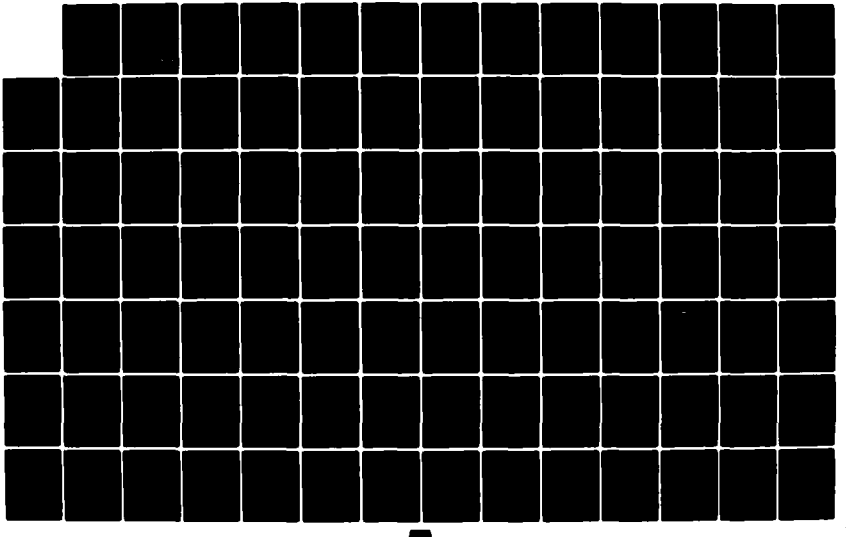
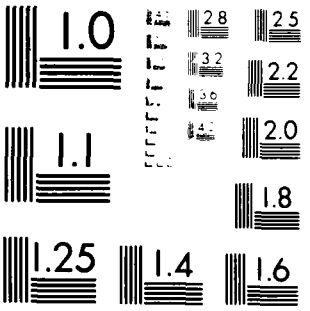
AD-A122 416 A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTROL APPLICATIONS(U) PRESRAY CORP PALO ALTO CALIF 30 JAN 78 CCA-78-03 N00039-77-C-0074 1/2

UNCLASSIFIED F/G 9/2 NL





A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 2

(1)

Technical Report
CCA-78-03
January 30, 1978

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

A Distributed Database Management System for Command and Control Applications

SEMI-ANNUAL TECHNICAL REPORT II

July 1, 1977 to December 31, 1977

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Naval Electronic System Command under Contract No. N00039-77-C-0074. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

Project Staff Members:

P. BERNSTEIN

S. FOX

N. GOODMAN

M. HAMMER

T. LANDERS

C. REEVE

J. ROTHNIE

D. SHIPMAN

1. Introduction

This report summarizes the second six months of a project entitled, "A Distributed Database Management System for Command and Control Applications", which has been undertaken by CCA and sponsored by ARPA-IPTO. The primary focus of this effort is to design and implement a distributed database management system called SDD-1 (System for Distributed Databases). SDD-1 is specifically oriented toward command and control applications and will be installed in phases and tested in the Advanced Command and Control Architectural Testbed (ACCAT) at the Naval Ocean Systems Center (NOSC) in San Diego.

The motivations for building and using any distributed data base system (and SDD-1 in particular) include the following:

1. reliability/survivability. By distributing a database and the database management function over a number of sites, the database can be made resistant to the failure or inaccessibility of any single site. This is a level of robustness that has no counterpart in a single-site system.

2. efficiency. By maintaining multiple copies of the same data, and by distributing them appropriately, the system should be able to provide a higher level of performance than could be achieved by a single-site system. This level of efficiency is due to the inherent parallelism in the system and the ability to distribute data geographically in such a way that it is stored near where it is frequently used.

3. modular growth. A distributed system can be designed to accommodate in a graceful fashion the growth of a database past local storage capacities as well as the introduction of additional sites into the system to cope with increased processing requirements.

Three important technical problems that must be addressed by a distributed DBMS and have been the focus of the SDD-1 design are:

- updating redundantly stored data;
- efficient distributed query processing; and

- achieving reliable operation.

The design of SDD-1 was essentially completed during this reporting period and implementation of the initial version was begun. The major design results reported in the previous technical report have been further refined and the following accomplishments have been achieved:

1. The redundant update technique has been generalized to incorporate the case where all data is not fully redundant and the method has been proved correct. This mechanism involves preanalysis of anticipated transactions to identify potential conflicts at database design time. This often permits transactions to avoid excessive synchronization and thus speeds up most updates by at least an order of magnitude.

2. The distributed query processing algorithm has been extended and generalized. The current algorithm incorporates a branch and bound technique to avoid choosing a poor final site to execute the query and the algorithm now handles disjunctions in a more optimal way. This algorithm is currently being implemented as part of the initial release of SDD-1.

3. The reliability mechanisms have been refined and further detailed. These mechanisms provide means for reliably broadcasting messages to multiple sites and for reliably delivering messages to sites even when they are down. The degree of reliability is "tunable" so that a particular application of SDD-1 can trade-off overhead with level of reliability.

Summaries of these design results are presented in section 2 of this report. Section 2.1 overviews the design and sections 2.2 through 2.4 elaborate the redundant update technique, distributed query processing algorithm and the reliability mechanisms.

The other activities of this project have dealt with an ongoing effort to provide Datacomputer services and improvements to the ACCAT. These have included:

1. A precompiled request feature was added to the Datacomputer to avoid most of the compilation overhead in frequently executed requests. The first version of this feature has been installed since September, 1977 and the second version will be installed at the end of January, 1978.

2. A mechanism for assigning priorities to different Datacomputer users was implemented. This mechanism enables high priority jobs to preempt other Datacomputer jobs until they have finished running. This priority scheduling facility was installed in September, 1977.

3. A boolean alerting feature that provides database alerting based on booleans dealing with both old and new values of fields in updated records. has been implemented and will be delivered at the end of January, 1978.

4. Consultation by CCA personnel has been provided to NOSC to assist in the smooth running of the Datacomputer.

These activities will be described in more detail in section 3 of this report.

As part of the SDD-1 design effort, CCA has made a concerted effort to publicize the results that have been achieved during this effort. Section 4 lists publications and meetings where SDD-1 has been discussed.

2. SDD-1 Design

The SDD-1 system consists of a network of modules that communicate over assorted communication channels. The distribution and redundancy of data in SDD-1 is invisible to users. A user connecting to any module is presented with the illusion that a complete, non-redundant database is resident at that module. In processing a user's transaction, SDD-1 assumes the responsibility for locating and accessing all data items referenced by the transaction and for updating in a consistent fashion all copies of the data items modified by the transaction. The overall architecture of SDD-1 is described in more detail in [ROTHNIE and GOODMAN a] and [CCA].

2.1 Overview of Design

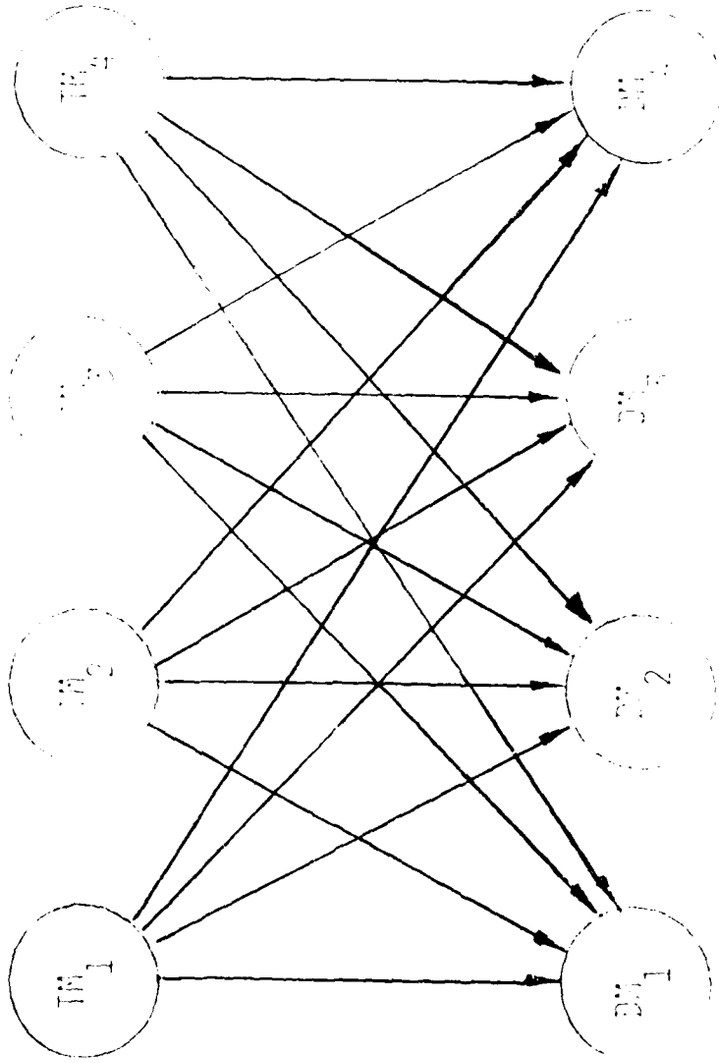
The internal architecture of the SDD-1 system is that it is composed of two kinds of modules: transaction modules (TM's) and data modules (DM's). Each site in the system may contain one or the other or both of these modules. All data is permanently stored at the DM's. A DM basically performs the functions of a local data base manager; it responds to commands issued it by TM's. It also performs computation on data items and may exchange data with other DM's. All of a DM's actions are performed at the direction of a TM. The TM's are responsible for processing user transactions, translating from the user's view of the data to the realities of its distribution and redundancy. The TM decides which copies of the logical data items requested by the transaction ought to be retrieved; obtains them by means of issuing commands to the appropriate DM's; causes the necessary computation to be performed on them, again by issuing instructions to DM's; and coordinates the updating of all data items to be modified. The TM essentially coordinates and supervises the execution of a transaction; the actual

labor is done by the DM's. Figure 2.1 illustrates the TM-DM configuration and shows the communication paths.

In SDD-1, a user's view of the data base is that provided him by a relational schema [CODD] for it; this schema, of course, contains no information regarding the distribution or redundancy of the data. A transaction is expressed as a program written in Datalanguage, a semi-procedural data manipulation language, that refers to the file and field names defined in the schema. Each TM may use a different mapping in translating references to these logical data items into accesses to physical data items stored at particular DM's. These mappings are referred to as materializations. (Observe that a transaction need not read all its data from a single DM.) A module of the TM, called the access planner, is responsible for determining the most efficient way of performing a computation that uses data items stored at different sites; it produces a sequence of move instructions and computation sequences, to be performed by the DM's involved in the transaction, to produce the transaction's results. These results include both output to the user and updates to the data base. The access planner is described in more detail in [WONG] and [REEVE et al] and is the subject of section 2.3.

SDD-1 Module Configuration

Figure 2.1



SDD-1 Architecture Overview

Two fundamental concepts underlie SDD-1: timestamps and protocols. Each TM in the system is equipped with a local clock that operates asynchronously with respect to all the others. Each transaction that is submitted to a TM is assigned a (system-wide unique) timestamp, reflecting the time of its execution. Furthermore, each variable stored at each DM has associated with it a timestamp value, which corresponds to the timestamp of the last transaction to have modified it. That is, each time a transaction instructs a DM to update the value of some variable x stored at that DM, the DM will examine t(x), the value of the timestamp associated with that variable. If t(x) is less than t(Tr), the timestamp of the transaction, then x is updated and t(x) is set equal to t(Tr); otherwise, the update is simply not performed. This mechanism is designed to allow the appropriate sequencing of transactions that are run concurrently at different TM's, even if they encounter different communication delays in transmitting updates to the same DM. Observe that two copies of the same variable stored at different DM's will not necessarily have the same timestamp value at all times.

Every transaction run at a TM operates under an intermodule coordination procedure called a protocol. The function of the protocols is to assure that transactions

running at different TM's do not interact in harmful ways that result in data base inconsistencies. Each transaction has associated with it a fixed protocol, which is determined by a preanalysis of all the kinds of transactions that may be performed against the database. This preanalysis determines the possible harmful interactions between these transactions and assigns them protocols so as to forestall these possibilities. The essential idea underlying the protocols is that a transaction may be forced to abort part way through its execution if it detects a situation which may potentially lead to a database inconsistency. If the protocol allows the transaction to run to completion, then no inconsistencies can result. A transaction that is aborted by its protocol will have to be resubmitted for execution at a later time; the protocols are so defined that the transaction will eventually be able to run. More detail on the redundant update technique used in SDD-1 appear in [ROTHNIE et al], [BERNSTEIN et al a] and [BERNSTEIN et al b] and are highlighted in section 2.2.

2.1.1 Distributed Data Organization

Each relation in SDD-1 has one domain named "tuple identifier" (TID) which is a key of the relation; that is, no two tuples of a relation can have identical TID values. Each relation is partitioned into a set of logical fragments. Logical fragments are defined by first partitioning the set of all possible tuples of the relations into a set of mutually exclusive partitions. For example, the EMPLOYEE relation could be partitioned by DEPARTMENT, so that each partition contains all of the employee tuples in a single department. A logical fragment consists of a projection of a partition on the TID domain and a unique subset of the other domains. The inclusion of the TID domain guarantees that the logical fragment has exactly one tuple for each tuple of the partition from which it was selected.

A stored copy of a logical fragment is called a stored fragment. Stored fragments are the units of data distribution; a stored fragment is either entirely present or entirely absent at a data module.

We do not require that two stored copies of a logical fragment at two different data modules be identical at all times. The redundant update mechanism will be responsible for only allowing consistent copies to be read (see section 2.2).

Each logical fragment is partitioned into logical data items, a stored copy of which is called a stored data item. A data item is the smallest updatable unit in the database.

Each logical data item may have several associated stored data items. Hence, when referencing a logical data item, it is necessary to choose a particular stored data item to reference. The concept of materialization is convenient here. Formally, a materialization is a total function from the set of logical fragments into the set of stored fragments. That is, a materialization is an assignment of a stored fragment for each logical fragment.

Each transaction is said to run in a particular materialization of the database. The materialization of a transaction specifies which copies of logical fragments are to be read. In order to maintain the internal consistency of all stored copies of a particular logical fragment, a transaction must perform its updates on all stored copies of each logical data item (not just the copy specified by the materialization).

2.1.2 Transactions

The basic unit of a user computation in SDD-1 is the transaction. Transactions are structured to execute in three sequential steps:

1. The transaction reads a subset of the database, called its read-set, into a workspace.

2. It does some computation on the workspace.

3. The transaction writes some of the values in its workspace back into a subset of the database, called its write-set.

The read-set and write-set of a transaction are defined on the logical database. That is, the transaction references only logical data items; it has no knowledge of its materialization or of the distribution and redundancy of stored copies.

The workspace into which data is read is, in general, distributed. That is, various parts of the workspace may reside at different data modules. In SDD-1, the execution

of a transaction is also, in general, distributed; processes running at various data modules operate on the portion of the workspace located at that data module.

To process a transaction, a transaction module must obtain the read-set data for the transaction's input and later write its output into copies of its write-set. These functions are performed by sending READ and WRITE messages, respectively, to data modules.

A READ message for a transaction is sent to a data module and is a request to read some of the stored data items at that data module. Each stored item that is requested must be the particular stored copy of a logical data item in the read-set of the transaction that is specified by the materialization in which the transaction runs. So, if a transaction wants to read logical data item x, and the transaction's materialization associates x with its particular stored copy at data module alpha, then to read x the transaction must send a READ message to alpha.

A WRITE message is sent from a transaction module to a data module to report updates that have taken place to certain data items as a result of executing a transaction by that transaction module. If a transaction updates a particular logical data item x, WRITE messages are sent to all data modules that have a stored copy of x (not just to

the one stored copy associated with the transaction's materialization).

A transaction module sends at most one READ message and at most one WRITE message to any particular data module on behalf of a single transaction. If a transaction reads data from two stored fragments which reside at the same data module, for example, then only one READ message will be issued to read from both fragments. This is an important point, as each data module must perform READ's and WRITE's as atomic operations; for example, none of the data read by a READ message can be updated by some WRITE while the READ is being processed.

2.2 Redundant Update

In this section, the SDD-1 method of updating redundantly stored data is described. The approach to redundant update described in this section attempts overcome the problems of update bottlenecks and heavy communications traffic by preanalyzing those transactions that will run frequently, so as to select those transaction types that can run using little or even no synchronization.

The preanalysis technique determines, for each type of transaction, the level of synchronization required for that transaction type. The analysis is based on knowledge of which portions of the database each transaction will read or write. The major assumption is that the types of transactions that account for most of the database activity are predictable in the sense that they only operate on certain restricted portions of the database.

2.2.1 System Consistency Guarantees

One of the important advantages of SDD-1 is its ability to maintain multiple copies of the same logical piece of data at several different data modules. It is this capability of SDD-1 that presents the most difficult technical problems. The system must maintain the consistency of all copies of data and ensure that the READ requests for a transaction retrieve a correct state of the database. In addition, transactions reading or writing data in several data modules must be synchronized to ensure that a transaction does not read partial results of another transaction. If transactions are allowed to run in an arbitrary interleaved manner without coordination, various anomalies in system operation may occur. The system design guarantees two properties which prevent these anomalies from occurring.

System Property 1: Convergence - If updates were to be quiesced, then after some finite period of time all transactions which read the same logical data item will retrieve the same value for it. Essentially this means that all physical copies of a logical data item will eventually converge to the same value.

System Property 2: Serial Reproducibility (or Serializability) - The operation of the system when running transactions in an interleaved manner is equivalent to a history of operation in which each of the transactions runs alone to completion before the next one begins. That is, the interleaved operation is reproducible by an equivalent one in which the transactions run serially. By "equivalent", we mean that each transaction produces the same output values and that the final state of the database is the same. The concept of serial reproducibility is crucial to an understanding of the system and will be taken up in detail later.

2.2.2 Timestamps

System property 1, convergence, is provided in SDD-1 through the use of the timestamping mechanism. Each TM has a clock used for generating globally unique timestamps. After a clock has been read, it cannot be read again until it has been incremented. By appending the TM number as the low order bits of each timestamp, we ensure that every timestamp is globally unique within the system. This method of generating unique timestamps was suggested in [THOMAS].

Each transaction, before being run, is assigned a unique timestamp. The transaction's timestamp will be carried on all its WRITE messages.

In addition, timestamps are maintained for every updatable physical data item in the database. Note that a timestamp is associated with each physical data item, rather than with the logical data item; there may be many physical copies of a logical data item and each copy of the logical data item has its own timestamp. This timestamp is the timestamp of the last WRITE message which updated that physical data item.

In order to implement property 1, convergence, each data module obeys the following rule: A data item is updated by a WRITE message if and only if the data item's timestamp is less than the timestamp of the WRITE message.

2.2.3 Interleaved Transactions

The system usually has many transactions in progress at any one time, both because there are multiple TM's operating concurrently within the system and because individual TM's are processing transactions concurrently. The resulting arbitrary interleavings of READs and WRITEs can introduce serious problems of database consistency. System Property 2, serial reproducibility, deals with this problem.

The issue of serial reproducibility arises because a system's atomic actions are at a finer granularity than its users' atomic actions. In our case, the users' atomic operations are user transactions, while the system's atomic actions can be taken to be the execution of READ and WRITE messages at the DM's. Each DM behaves as if READ's and WRITE's are processed as indivisible units. That is, it is not possible for a READ operation to observe the effects of only a part of a WRITE operation at a DM.

When a system allows the execution of several user transactions at the same time, then the system atomic

operations corresponding to different user transactions are interleaved. There is no guarantee that the behavior of such a system conforms to the user's expectation that each transaction is treated as an indivisible unit (a user's transaction should not examine the database during the execution of another user's transaction, when the database is possibly in an inconsistent state).

Serial reproducibility requires that a system operating in an interleaved manner is equivalent to a system in which each transaction is processed in its entirety before another one is begun. In other words, for any given interleaved execution, there exists an ordering of atomic transactions, called a serial ordering, which is equivalent to the interleaved operation which in fact occurs. By "equivalent" we mean that each transaction in the interleaved ordering reads the same data as it would have read if the transactions had been run one at a time in the serial order (and hence, will produce the same output).

If a system does not guarantee serial reproducibility then anomalies can result from operation of the system. Consider, for example, the following scenario in SDD-1. We assume a single copy of data item x, which initially has the value x=0. There are two transactions in the

system; transaction i sets x:=x+1, and transaction j sets x:=x+2. The following sequence of events occurs:

Transaction i reads x=0

Transaction j reads x=0

Transaction j sets x:=2

Transaction i sets x:=1

Any execution of the two transactions one after the other would have resulted in setting x to 3. The result of the interleaved execution was to set x to 1, contrary to the user's intention. To guarantee serial reproducibility, we need a mechanism that prevents these kinds of undesirable interleavings.

2.2.4 Transaction Classes

The problem of interleaved transactions is not unique to distributed systems. Numerous solutions have been devised for non-distributed systems, most notably locking mechanisms. These techniques do not, however, generalize well to distributed systems. A number of proposals have been suggested for extending locking mechanisms to distributed systems that contain redundant data. These techniques are reviewed in [ROTHNIE and GOODMAN b]. We feel, however, that such techniques require unacceptably large amounts of network transmission and delay whenever there is considerable data redundancy.

Yet at first glance the network transmission seems to be necessary. How can one TM safely proceed to run a transaction without first consulting other TM's to determine that it does not interact badly with transactions currently executing elsewhere?

Our solution to this problem is to have the DBA establish a static set of transaction classes. Each transaction class is defined in terms of its logical read-set and write-set and is assigned to run at a particular TM. A

transaction can run in a class if the read-set and write-set of the transaction is contained (respectively) in the read-set and write-set of the class. Classes need not be disjoint, so a transaction may fit into more than one class. In this case, the decision as to which class should be chosen is made by the terminal module that accepts the transaction. The terminal module will normally choose a class that requires the least amount of synchronization, and is therefore the least expensive class (synchronization-wise) to use.

The predefined classes reflect the typical transactions that are intended to run at each site in the network. Since each TM is aware of the complete set of transaction classes assigned to foreign transaction modules, it can know exactly what potential conflicts its own transactions have with those that might be running at other TM's.

From the information contained in the class definitions, a TM can determine the degree and nature of coordination necessary to ensure a serially reproducible ordering of transactions. We believe that, for many kinds of applications, the most frequent determination will be that no coordination whatsoever is actually required to run a transaction. In such a case, the transaction is just immediately executed, since it does not interact badly

with transactions submitted elsewhere. In other cases, an analysis of the class definitions might indicate that the pending transaction could be involved in a potential conflict and some coordination is necessary with respect to particular foreign classes. Our purpose here is to develop a method of determining exactly what conflicts occur and to provide coordination mechanisms that eliminate the conflict.

If the problem of determining exactly what conflicts might occur required run-time calculations when each transaction was introduced at a class, then the concurrency control mechanism would potentially be quite expensive. Actually, since the class definitions are static, the computations checking for potential conflicts can be done once, when the class definitions are selected. Selecting the appropriate coordination mechanism at run-time amounts to a table look-up. So, the only significant run-time overhead is the coordination mechanism itself. If no coordination is found to be necessary, then the run-time overhead is negligible. This is in contrast to locking mechanisms which always set locks, whether or not the synchronization is really required.

2.2.5 Class Pipelining Rule

The first question to address is the issue of the serializability of transactions which execute in the same class. To ensure this, we require that within a class all of the transactions are actually executed serially, one after another. This is expressed as follows-

> Class Pipelining Rule: For any particular data module and transaction class, READ and WRITE messages from that class arrive and are processed in timestamp order.

The class pipelining rule forces transactions that run in a single class to be processed serially at all DM's in the same order. So, two transactions from a single class are never interleaved at a single DM nor are they processed by two DM's in two different orders. This is sufficient to guarantee noninterference of any two transactions that run in a single TM.

2.2.6 Class Conflict Graphs

Given the set of class definitions, we need to detect potentially harmful interactions between classes. The approach used to resolve these questions involves the construction and analysis of a class conflict graph.

A class definition specifies a logical read-set and write-set and a materialization. This is the only information required to determine class conflicts. From the read-set and the materialization, the READ messages needed by the class can be predicted. From the write-set, the WRITE messages needed by the class can be predicted, since a WRITE message must be sent to all copies of the logical write-set. Since all READ and WRITE messages are predictable, we will be able to predict all possible harmful interactions between classes.

A class is represented in the class conflict graph as three types of nodes connected by edges. The three types of nodes are e, r and w nodes.

An e node represents the execution of a transaction which runs in the class. A class superscript (e.g. e^{T_i})

designates the class identifier for the transaction class. (Throughout this report, transactions will be indicated by lower case letters, and transaction classes by lower case letters with an overscore.) The graph includes exactly one e node per class.

An r node represents the processing of a READ message to retrieve data for transactions in the class. A superscript represents the class identifier and a subscript indicates to which DM the READ message would be sent (e.g. $r^{\bar{j}}_{alpha}$ represents a READ message from a transaction in class $\bar{j}$ to DM_{alpha}). (Lower case Greek letters denote DMs.) For any class, there is one r node for each DM which stores part of the class's (physical) read-set.

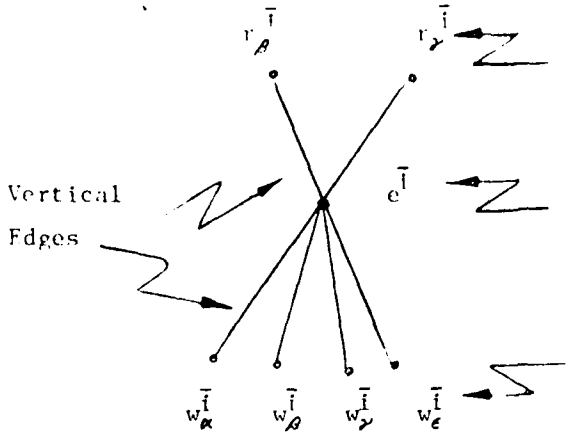
A w node represents the processing of a WRITE message issued by a transaction running in the class. Again, a superscript indicates the class identifier and a subscript indicates the DM to which the WRITE message would be sent (e.g. $w^{\bar{i}}_{gamma}$). For any class, there is one w node for each DM on which a copy of (some of) the write-set items lie.

Edges connect the e node for a particular class with the r and w nodes for that class. These edges are called <u>vertical</u> edges, because of the convention that, for each

class, r nodes are drawn above the e node and w nodes are drawn below the e node.

Figure 2.2 illustrates the representation of a class whose read-set lies in two datamodules and whose write-set lies on four datamodules.

After all the predefined transaction classes have been placed in the graph, additional edges are added to indicate interactions between the classes.



There are two READ messages, one to DM_α and the other to DM_γ

This is transaction class #14

Data must be written to four DM's: α β γ ϵ

Figure 2.2 Representing transaction classes in the graph

Where two classes have a read/write intersection, a diagonal edge is drawn. The edge is drawn between an r node which represents the reading of some particular physical data item and a w node which represents the writing of that same item (see Figure 2.3). Note that such a diagonal edge only connects r and w nodes with the same DM subscript, since a physical data item resides at only one DM. If the intersection of one class's read-set and another's write-set spans more than one DM, then several diagonal edges connect the two classes (see Figure 2.4).

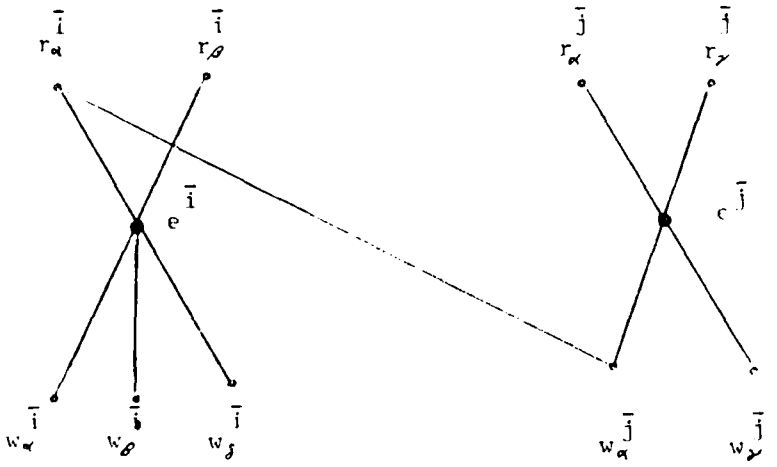


Figure 2.3 - The diagonal edge indicates that class $\bar{i}$ reads some data item from DM_α which can be written by class $\bar{j}$.

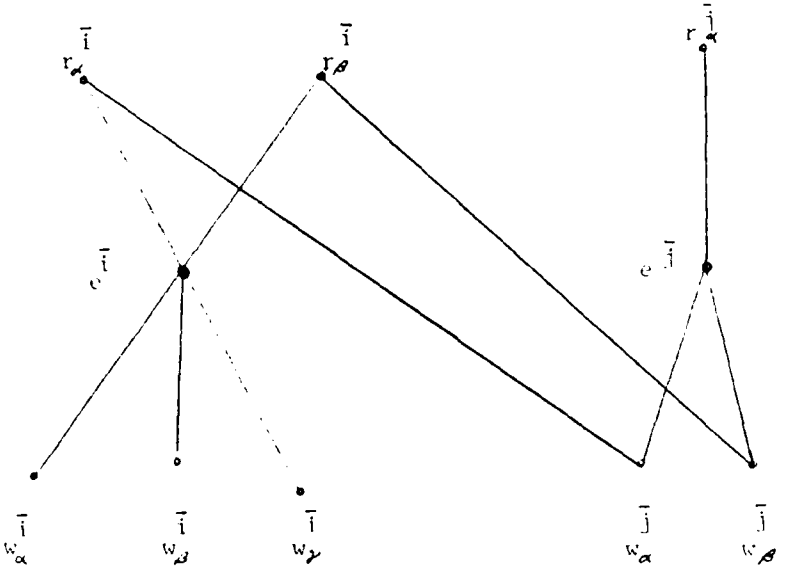


Figure 2.4 - Here two diagonal edges connect the two classes since the read/write intersection exists at both DM_α and DM_β

Horizontal edges are drawn between e nodes of two classes that have a logical write/write intersection (see Figure 2.5).

The graph must contain all classes and all possible vertical, diagonal and horizontal edges.

The conflict graph is used to determine unsafe interactions among a set of classes. By "unsafe" we mean

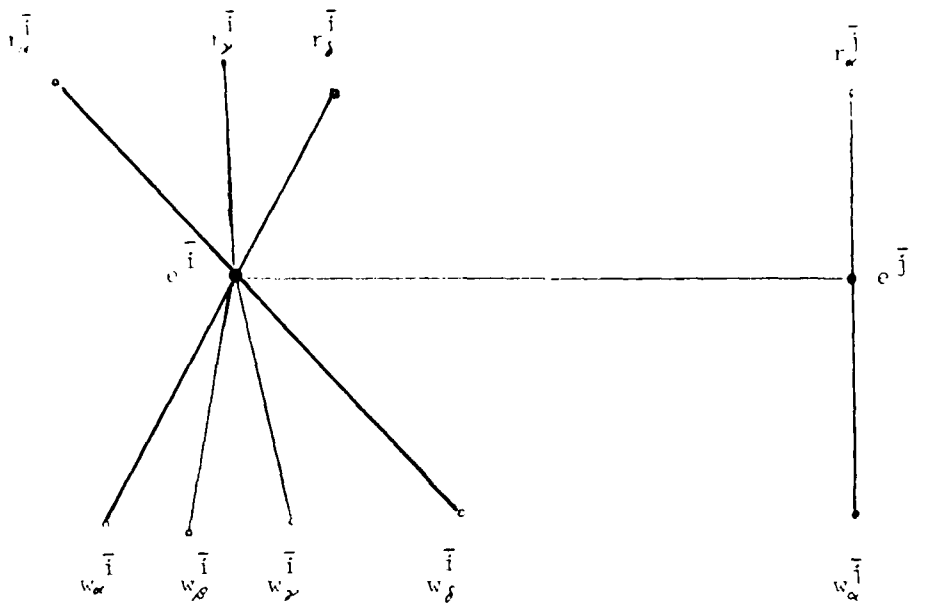


Figure 2.5 - A horizontal edge is added to the graph when two classes write the same data item.

that the classes can interact in such a way that there is no serial ordering of transactions that is equivalent to the interleaved execution that actually occurred. The interpretation of the diagonal and horizontal edges applied to a given interleaved execution is the key to determining transaction serializability.

2.2.7 Graph Cycles and Nonserializability

Suppose the system executes in a manner that permits the interleaving of READ and WRITE messages from different transactions. We call such an interleaved execution a log. If the execution is not interleaved, that is, if transactions execute serially one after the other, then we call the execution a serial log. Our goal is to only permit the system to produce logs that are serially reproducible. This means that for each log resulting from the execution of the system, there must exist a serial log that produces the same effect on the database. We say that two logs are equivalent if they produce the same effect on the database.

Of course if the transactions in a log are arbitrarily reordered into a serial log, the resulting serial log will not necessarily be equivalent to the given log. The conflict graph helps us to characterize precisely those serial logs that produce the same effect as a given log.

Consider diagonal graph edges. A diagonal edge represents a read/write intersection between two classes. If one transaction from each of the two classes appears in the

given log, then in any equivalent serial log the transactions should appear in the same relative order as their intersecting READ and WRITE messages were processed in the given log. For if the READ message of one transaction preceded the WRITE message of the other in the given log, but the transactions appear in the reverse order in the serial log, then in the serial log the READ message may read different values for some of its inputs in the serial log than reads in the given log. So, the transaction corresponding to the READ may produce a different output in the serial log than in the given log. That is, the two logs are not necessarily equivalent. This is just to say that only some serial reorderings of the given log are possible, given the existence of this diagonal edge. (Actually, the above claim about permissible serial reorderings is somewhat too strong, as shown in [PAPADIMITRIOU et al]. However, the reasons are quite technical in nature and are not needed to gain an understanding of the interpretation of conflict graphs.)

Consider classes $\bar{i}$ and $\bar{j}$ in figure 2.3. We denote READ and WRITE messages using a notation similar to that of node labels. The processing of the READ message for transaction i at DM_{alpha} is denoted R^{i}_{alpha}; the processing of the WRITE message for transaction i at DM_{alpha} is denoted W^{i}_{alpha}.

Assume two transactions, say i and j, are running concurrently in classes $\bar{i}$ and $\bar{j}$ respectively. If the READ message R^i_{alpha} is processed at DM_{alpha} before the WRITE message W^j_{alpha} is processed, then any equivalent serial ordering must have transaction i precede transaction j. This must be so, for otherwise transaction i would have read the results of the update made by transaction j. On the other hand, if the WRITE message W^j_{alpha} is processed before the READ message R^i_{alpha}, then transaction j must precede transaction i.

To reiterate, a diagonal edge implies a particular relative ordering in any serial log that is equivalent to the given interleaved execution. The particular ordering that is chosen depends on the particular order in which READ and WRITE messages were processed; however the relative serial ordering of transactions from classes with a diagonal edge connecting them is not arbitrary.

Horizontal edges also affect possible reorderings of transactions. A horizontal edge indicates an intersection of write-sets. Whenever two transactions write the same data, the update from the transaction with the greater (i.e. later) timestamp takes precedence over the update from the transaction with the smaller (i.e. earlier) timestamp. If two transactions in different classes

appear in an interleaved execution and have a write/write intersection, then they must appear in timestamp order in any equivalent serial log. Otherwise, the effect of the intersecting write messages would be reversed, thereby producing a different database state. Notice that it is the timestamp order of the transactions and not the order in which the WRITE messages were processed that is significant here. This is because the rule by which WRITE messages are processed uses the timestamps, not the order of arrival of the WRITE messages, to determine which write operations are actually applied.

So, a horizontal edge also implies a particular relative ordering of certain transactions in any serial log that is equivalent to the given interleaved execution. This ordering is always the timestamp ordering of the transactions that have the write/write intersection.

In the same way that diagonal and horizontal edges restrict the ways in which transactions can be reordered without upsetting the resulting database state, paths of edges can restrict reorderings of transactions as well. For example, a particular diagonal edge may imply that transaction i must precede transaction j and an adjacent horizontal edge may indicate that transaction j must precede transaction k (see figure 2.6). So, the net

effect of this path is that transaction i must precede transaction k, even though no single edge may connect

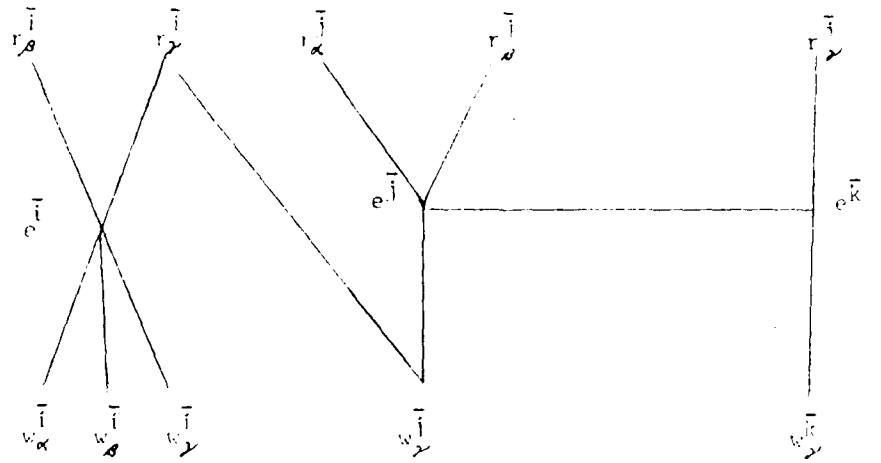


Figure 2.6 - A path between two classes in the graph indicates that transactions in these classes must be serialized in some particular order.

their respective classes in the conflict graph.

Now, suppose again that we have a conflict graph and a log of interleaved transactions. Suppose that for each pair of transactions, say i and j, the log and graph edges never imply both that i must precede j and that j must precede i in the serial reordering. That is, either i and j can appear in an arbitrary order, or there is only one order that will do. Then it is easy to see that there must be a serial log equivalent to the given log. Any

serialization that preserves the relative orderings that are demanded by the graph serves the purpose.

However, suppose instead that there are two transactions such that one path in the graph requires that they appear in one order and another path in the graph requires that they appear in the other order. Then there is no equivalent serial log that includes these two transactions, for whatever order that they appear in the serial log, the graph indicates that they must also appear in the other order. In this case, there are two different paths connecting the two transactions' classes in the graph. These two paths constitute a cycle in the graph. So, apparently a cycle in the graph corresponds to a non-serializable execution of transactions. If there are no cycles, then there is at most one path connecting any pair of classes. Hence, the graph can only require that two transactions be serialized one way or the other, but never both ways. So, a cycle-free graph implies that every log is serializable, and no synchronization whatsoever is required. The preceding informal argument demonstrating this fact will be proved quite rigorously in Section 4.

Consider the cycle in Figure 2.7 consisting of two diagonal edges and four vertical edges. If we examine a

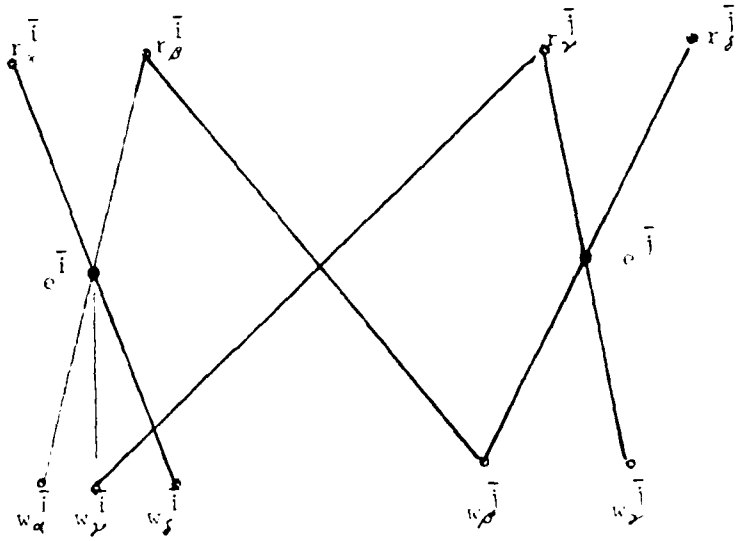


Figure 2.7 - Cycles represent situations in which non-serializability is possible.

case of concurrent transactions in each of the two classes and the particular sequence of events in which the READ message R^{i}_{beta} is processed before the WRITE message W^{j}_{beta}, and the READ message R^{j}_{gamma} is processed before the WRITE message W^{i}_{gamma}, then there is no serial ordering of the two transactions which is equivalent to their interleaved ordering. This follows because the $r^{\bar{I}}_{beta}$-$w^{\bar{J}}_{beta}$ edge requires that the transaction in class $\bar{I}$ occurs before the transaction in class $\bar{j}$, yet the $r^{\bar{J}}_{gamma}$-$w^{\bar{I}}_{gamma}$ edge implies the opposite relative ordering. Therefore, it must be the case that no equivalent serial ordering exists.

We have shown that potentially dangerous interleavings can be identified by a cycle in the class conflict graph. So, as long as no cycles exist, the class pipelining rule is sufficient to guarantee serializability. Where cycles do exist, some synchronization among classes is required. In SDD-1, this synchronization is accomplished by protocols.

2.2.8 Protocol P3

When a cycle exists in the conflict graph, then an interleaved execution might be such that a pair of transactions, i and j, must be serialized with i preceding j and j preceding i, clearly an impossibility. Protocol P3 prevents this situation by making the following guarantee: <u>If two transactions belong to two classes connected by a diagonal edge in a cycle, then the timestamp order of the two transactions is the same as the relative ordering dictated by the diagonal edge</u>. For example, suppose the edge $(r^{\bar{I}}_{alpha}, w^{\bar{J}}_{alpha})$ lies on a cycle and transaction i executes in class $\bar{I}$ and j executes in class $\bar{J}$. Then, assuming protocol P3 is observed, R^{i}_{alpha} is processed before W^{j}_{alpha} if and only if the timestamp of i is smaller that the timestamp of j. Before describing how P3 accomplishes this task, let us first examine how P3 prevents nonserializable executions.

Consider again transaction i and j above. Since they apparently must be serialized in both orders, there must be two independent paths connecting them in the graph, such that one path requires that i precede j and the other requires that j precede i. Suppose the timestamp of i is smaller than that of j. So, the path that requires j to precede i in the serial reordering is trying to serialize them in reverse timestamp order. But suppose every transaction pair connected by a diagonal edge in this path observes P3. Then each such pair must be serialized in timestamp order, as P3 requires. Consider a pair of transactions connected on the path by a horizontal edge. Following the discussion about horizontal edges in the last section, they too must be serialized in timestamp order. Thus, every pair of transactions in the interleaved execution that corresponds to a graph edge along this path must be serialized in timestamp order. The net effect (by induction on the length of the path) is that the entire path requires that i and j be serialized in timestamp order. But this is a contradiction, since the chosen path was one that required the transactions to be serialized in reverse timestamp order. The conclusion is that all paths in the graph between $\bar{i}$ and $\bar{j}$ require that i and j be serialized in timestamp order. Protocol P3 prevents the case that there are two independent paths

between $\bar{i}$ and $\bar{j}$ that require opposite relative orderings.

To implement protocol P3, we need to synchronize the READ and WRITE messages of transactions that correspond to the endpoints of a diagonal edge in a cycle. To explain the operation of P3, suppose that the edge ($r^{\bar{i}}_{alpha}$, $w^{\bar{j}}_{alpha}$) is a diagonal edge in a cycle; so, for each transaction i in class $\bar{i}$, R^{i}_{alpha} has to run P3 against class $\bar{j}$ at DM_{alpha}. This is accomplished by appending a read condition to each read message R^{i}_{alpha}. The read condition includes the timestamp of transaction i, say TS_i, and the name of the class against which P3 is being run, in this case $\bar{j}$. A data module, upon encountering a READ message with the attached read condition $< TS_i, \bar{j} >$, must not process the READ until it is certain that all WRITE messages from $\bar{j}$ with timestamps prior to TS_i have been received and processed, and that it has not processed any WRITE messages from $\bar{j}$ with a timestamp greater than TS_i. This ensures that the READ messages R^{i}_{alpha} is processed before a WRITE message from $\bar{j}$ if and only if TS_i is smaller than the timestamp of the transaction corresponding to the WRITE message. That is, it guarantees that the diagonal edge forces transactions from the two classes to be serialized in timestamp order. We refer to this mechanism as protocol P3, and would say, for example, that transactions in class $\bar{i}$ run protocol P3 against transactions in class $\bar{j}$ at DM_{alpha}.

Several problems arise about the operation of protocol P3. Suppose the DM has already processed a WRITE message from the specified class $\bar{j}$ with a timestamp greater than TS_i. In this case, the READ message must be rejected by DM_{alpha}. The initiating TM then assigns a new timestamp to the transaction and resubmits its READ requests. Notice that all READ messages must be resubmitted if any READ message is rejected.

A more serious problem is how to guarantee that a DM has received all WRITE messages through some particular time. The solution lies in the class pipelining rule. Recall that READ and WRITE messages from a class to a DM must be processed in timestamp order. If DM_{alpha} wants to process all WRITE messages from $\bar{j}$ up to but not past time TS_i, it simply processes all WRITE messages from $\bar{j}$ until it receives one with a timestamp greater than TS_i. It holds this WRITE message until R^i_{alpha} is processed, thereby satisfying the read condition attached to R^i_{alpha}.

Unfortunately, if class $\bar{j}$ is idle because it has no transactions to process, DM_{alpha} may need to wait for a long time until a message timestamped later than TS_i arrives from $\bar{j}$. To handle this problem we have TM's send out NULLWRITE messages to appropriate DM's. A NULLWRITE message specifies a class and a timestamp. It is

semantically equivalent to a WRITE message that does not update any data. When a DM receives such a NULLWRITE message, it can be sure that it has received all WRITE messages from the indicated class through the given timestamp.

TM's will send out NULLWRITEs on a periodic basis. In addition, a TM may be specifically requested to send a NULLWRITE for a particular class and timestamp. This specific request is in the form of a SENDNULL message and may be sent by either another TM or a DM. A discussion and analysis of various strategies for sending NULLWRITE and SENDNULL messages will appear in a later report.

To illustrate the use of protocol P3 for eliminating bad interleaved executions, let us reconsider the anomalous scenario discussed in section 2.2.3, this time adding a bit more structure to the problem.

We assume a single copy of data item x, residing at DM_{alpha}, with initial value x=∅. Class $\bar{i}$ has been defined to run at TM_{alpha} with read-set = {x} and write-set = {x}. Class $\bar{j}$ has been defined to run at TM_{beta} with read-set = {x} and write-set = {x}. The class graph in this situation is shown in figure 2.8. Notice that a cycle is present and that transactions in class $\bar{i}$ must run P3 against class $\bar{j}$ and that transactions in class $\bar{j}$ must run P3 against class $\bar{i}$.

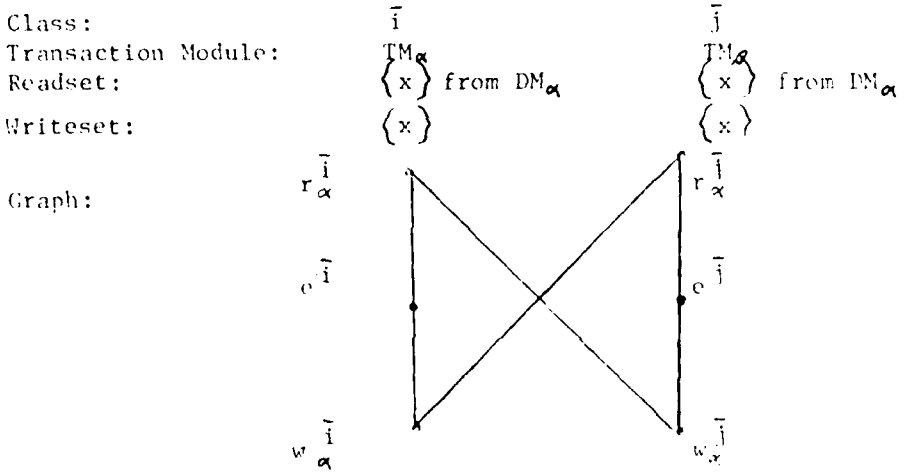


Figure 2.8 - Class Conflict Graph for Example in Section 2.2.8

A transaction, i, arrives at TM_{alpha} of the form x:=x+1. TM_{alpha} assigns the transaction to class $\bar{i}$ and gives it timestamp TS_i. A transaction, j, arrives at TM_{beta} of the form x:=x+2. TM_{beta} assigns the transaction to class $\bar{j}$ and gives it timestamp TS_j. TS_i and TS_j cannot be equal because all timestamps in the system are unique. Let us assume that $TS_j < TS_i$. Now the following sequence of events occurs:

1. TM_{alpha} sends a READ message, R^{i}_{alpha}, to DM_{alpha} to retrieve the value of data item x for transaction i. This READ includes a P3 read condition against class $\bar{j}$. The READ message cannot be immediately processed because WRITE messages through time TS_i from class $\bar{j}$ have not yet been received at DM_{alpha}.

2. TM_{beta} sends a READ message, R^{j}_{alpha}, to DM_{alpha} to retrieve the value of data item x for transaction j. The READ message can be immediately processed (the presence of a class $\bar{i}$ READ message at DM_{alpha} with timestamp $TS_i > TS_j$ insures that all WRITE messages from class $\bar{i}$ have been received through time TS_j). The result of the READ is x=∅.

3. TM_{beta} sends a WRITE message for transaction j to DM_{alpha} setting x:=2.

4. TM_{beta} sends a NULLWRITE message to DM_{alpha} with timestamp $TS_{j'} > TS_i$. (This message may be a response to a SENDNULL request from TM_{alpha}. The class pipelining rule requires that this message could not be sent before the WRITE message with time $TS_j < TS_{j'}$). The READ message for transaction i can now be processed. (The presence of the NULLWRITE message at DM_{alpha} with timestamp $TS_{j'} > TS_i$ satisfies the P3 read condition.) The result of the READ is x=2.

5. TM_{alpha} sends a WRITE message for transaction i to DM_{alpha} setting x:=3. Notice that this WRITE message overwrites the earlier value of x=2 because the earlier value was associated with timestamp TS_j and the current WRITE message has timestamp $TS_i > TS_j$.

The final value of data item x is 3, as expected. The anomalous interleaving that was described in the example of section 2.2.3 has been prevented by the use of protocol P3.

We have seen that by locating graph cycles, by finding every class that lies at the r-end of a diagonal edge embedded in a cycle, and by having transactions in that class run protocol P3, we can guarantee that all interleaved executions will be serializable. However, there are situations in which weaker protocols (i.e., protocols that allow more concurrency) than P3 may be used. This leads us to a discussion of protocols P2 and P2f.

2.2.9 Protocol P2

The main opportunity for weakening the P3 protocol arises in connection with the transactions that participate in a conflict graph only with their read-nodes. These read-only transactions contribute to non-serializability only because they may observe certain WRITE messages being processed in reverse timestamp order. For example, suppose we have classes $\bar{i}$, $\bar{j}$, and $\bar{k}$ connected by the edges ($w^{\bar{i}}_{alpha}$, $r^{\bar{j}}_{alpha}$) and ($r^{\bar{j}}_{alpha}$, $w^{\bar{k}}_{alpha}$) as shown in figure 2.9. Class $\bar{j}$ is a read-only transaction whose read-set intersects the write-sets of classes $\bar{i}$ and $\bar{k}$. Suppose transactions i, j, and k execute in classes $\bar{i}$, $\bar{j}$, and $\bar{k}$ (respectively) such that k is timestamped before i which is timestamped before j. At DM_{alpha}, the following sequence of events might occur: first W^{i}_{alpha} is processed, then R^{j}_{alpha} is processed, then W^{k}_{alpha} is processed. In this case, even though k is timestamped earlier that i, from j's point of view transaction i precedes transaction k, since it sees i's update but has not yet seen k's update. That is, this interleaved execution requires that transaction i be serialized in front of transaction k, which is the reverse timestamp

order. If another path in the conflict graph connected $\bar{i}$ to $\bar{k}$ such that the interleaved execution required the timestamp ordering, then the impossible requirement that i both precede and follow k in the serial reordering means that the execution is not serializable. In the previous section we showed that if R^j_{alpha} ran P3 against $\bar{i}$ and $\bar{k}$ (due to the two diagonal graph edges), then this non-serializable situation could not arise. However, there is a weaker protocol that R^j_{alpha} can run in this

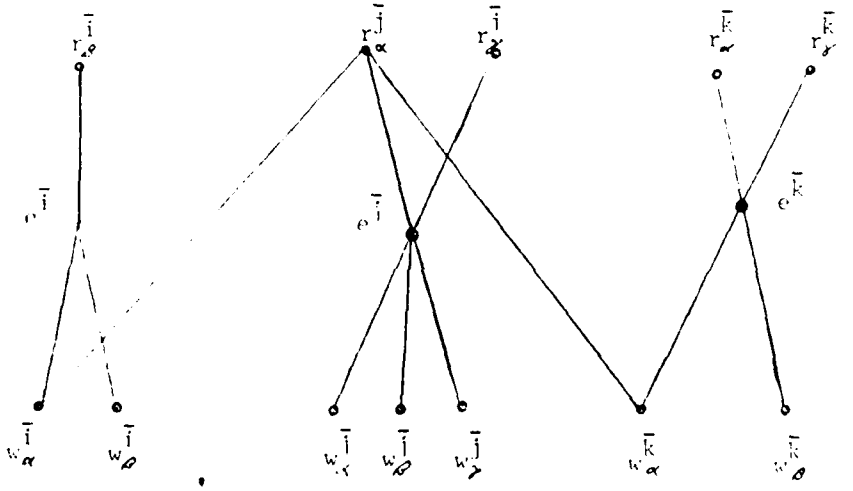


Figure 2.9 - A transaction reading from two other transaction classes may force a relative ordering of these classes' transactions in equivalent serial orderings.

situation that has the same effect.

The effect we want to produce is that if W^i_{alpha} is timestamped after W^k_{alpha} and W^i_{alpha} is processed before R^j_{alpha}, then W^k_{alpha} is processed before R^j_{alpha} as well. If this condition is made to be true (by some protocol) then R^j_{alpha} cannot observe W^i_{alpha} and W^k_{alpha} to execute in reverse timestamp order. The protocol that has this effect is called P2.

Protocol P2 applies to a read message R^j_{alpha} if and only if there are classes $\bar{i}$ and $\bar{k}$ such that ($w^{\bar{i}}_{alpha}$, $r^{\bar{j}}_{alpha}$, $w^{\bar{k}}_{alpha}$) is a subpath in a cycle in the conflict graph (where j runs in class $\bar{j}$). In this case, we say that R^j_{alpha} must run protocol P2 against classes $\bar{i}$ and $\bar{k}$ at DM_{alpha}. If protocol P2 is used, then R^j_{alpha} need not run protocol P3 against $\bar{i}$ and $\bar{k}$, as would normally be indicated by the diagonal edges. Since P2 prevents R^j_{alpha} from observing transactions in $\bar{i}$ and $\bar{k}$ in reverse timestamp order, R^j_{alpha} will not interfere with serializing transactions in $\bar{i}$ and $\bar{k}$ in timestamp order, as desired.

To run R^j_{alpha} under P2 against $\bar{i}$ and $\bar{k}$, DM_{alpha} must ensure that, at the time R^j_{alpha} is processed, there is a timestamp TS_o, such that all WRITE messages from classes $\bar{i}$ and $\bar{k}$ whose timestamps are less than TS_o have been

processed at DM_{alpha}, and no WRITE messages from classes $\bar{i}$ and $\bar{k}$ whose timestamps are greater that TS_o have been processed. The specific timestamp, TS_o, is not given by the READ message R^j_{alpha} but rather is selected by DM_{alpha}. As long as there exists some TS_o through which WRITEs from the classes $\bar{i}$ and $\bar{k}$ have been processed but beyond which they have not been processed, then R^j_{alpha} will only be able to observe transactions in classes $\bar{i}$ and $\bar{k}$ to have been run in their relative timestamp order.

The implementation of protocol P2 requires an extension to the read condition mechanism. Since the DM is expected to choose a convenient TS_o (cf. P3 where the timestamp is prespecified in the READ message), the timestamping in the read condition cannot be determined until the READ message is processed. So, a named timestamp marker may be supplied in place of a particular timestamp in the read condition. Whenever a DM encounters a timestamp marker in a read condition, it may choose an appropriate time itself, with the proviso that when two or more read conditions are given for a single READ message, all timestamp markers with the same name must be assigned the same timestamp value.

For R^j_{alpha} to run P2 against classes $\bar{i}$ and $\bar{k}$, R^j_{alpha}'s READ message must include two read conditions, <TSM, $\bar{i}$>

and <TSM, $\bar{j}$>, where TSM is a timestamp marker. By satisfying the read conditions, DM_{alpha} fulfills the protocol P2 condition against classes $\bar{i}$ and $\bar{k}$, as desired.

It is interesting to note that protocol P2 is strictly weaker than P3 in the following sense. If R^{j}_{alpha} runs P3 against classes $\bar{i}$ and $\bar{k}$ at DM_{alpha}, then R^{j}_{alpha} satisfies the P2 constraint against $\bar{i}$ and $\bar{k}$ as well. The converse is not true. Since P2 always permits more concurrency than P3, it is always advantageous to run P2 in place of P3 where ever possible.

An example will illustrate the use of protocol P2. Suppose there are two data items of interest, x and y, which reside at both DM_{alpha} and DM_{beta}; initially x=0 and y=0. We assume there is an integrity constraint requiring that $y \leq x^2$. Three classes have been defined. Class $\bar{i}$ runs at TM_{alpha}, reads x from DM_{alpha} and writes x. Class $\bar{j}$ runs at TM_{alpha}, reads x from DM_{alpha} and writes y. Class $\bar{k}$ runs at TM_{beta}, and reads x and y from DM_{beta}. A class conflict graph for this configuration is shown in figure 2.10. Notice that a cycle is present and that transactions in class $\bar{j}$ must run P3 against transactions in class $\bar{i}$ at DM_{alpha} and that transactions in class $\bar{k}$ must run protocol P2 against classes $\bar{i}$ and $\bar{j}$ at DM_{beta}.

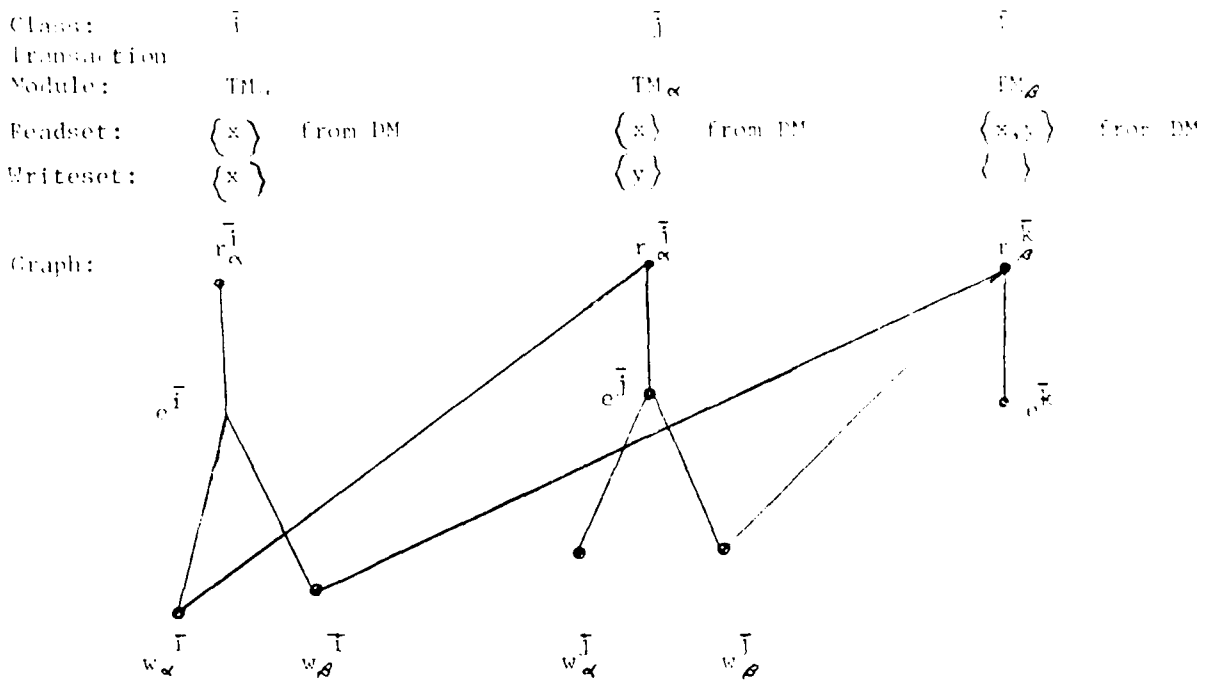


Figure 2.10 - Class Conflict Graph for Example in Section 2.2.9

Transaction i is received at TM_{alpha}, requests to perform the computation x := x+1, is assigned to class $\bar{i}$, and is given timestamp TS_i. Transaction j is received at TM_{alpha}, requests to perform y := x^2, is assigned to class $\bar{j}$, and is given timestamp $TS_j > TS_i$. Transaction k is received at TM_{beta}, requests to print the values of x and y on the user's terminal, is assigned to class $\bar{k}$, and is given timestamp $TS_k > TS_j$. Notice that each of these transactions preserve the constraint that $y \leq x^2$. No

serial ordering of the transactions could invalidate this condition.

First, we consider an anomalous scenario in which transaction k does not run protocol P2 as is required:

1. TM_{alpha} sends a READ message to DM_{alpha} for transaction i and retrieves x=0.

2. TM_{alpha} sends WRITE messages to DM_{alpha} and DM_{beta} for transaction i. Each WRITE message contains timestamp TS_i and the assignment x := 1.

3. DM_{alpha} processes the WRITE f r transaction i (but DM_{beta} has not yet done so).

4. TM_{alpha} sends a NULLWRITE message for class $\bar{I}$ with timestamp $TS_{i'} > TS_j$ to DM_{alpha}.

5. TM_{alpha} sends a READ message to DM_{alpha} for transaction j and retrieves x=1. (The P3 read condition on this READ message is immediately satisfied because of the previously received NULLWRITE message.)

6. TM_{alpha} sends WRITE messages to DM_{alpha} and DM_{beta} for transaction j. Each WRITE message contains timestamp TS_j and the assignment y := 1.

7. DM_{alpha} processes j's WRITE message.

8. DM_{beta} processes j's WRITE message.

9. TM_{beta} sends a READ message to DM_{beta} for transaction k, retrieving x=0, y=1.

10. Transaction k prints x=0, y=1 on the user's terminal.

11. DM_{beta} processes the WRITE message from i, thereby setting x=1.

The user has seen an impossible state of the database (i.e., x=0, y=1) printed by transaction k, with $y > x^2$. The problem is that k is reading both the input and output of another transaction, j. However, k is reading the new value of the output but an old value of the input on which that output is based.

If k had run protocol P2 as required, then this situation could not have occurred. By replacing steps (9)-(11) with the following, we obtain a correct scenario in which k satisfies P2.

9. TM_{beta} sends a READ message to DM_{beta} for transaction k. The P2 read condition requires that WRITE's from classes $\bar{i}$ and $\bar{j}$ be processed through some common time. Now $\bar{j}$ has been processed through

time TS_j but class $\bar{i}$ has not been processed through that time yet.

10. DM_{beta} processes the WRITE message for i.

11. A NULLWRITE message arrives at DM_{beta} for class $\bar{i}$ with timestamp TS_i ->TS_j.

12. DM_{beta} can now process the READ message from k, since WRITE's from both $\bar{i}$ and $\bar{j}$ have been processed through time TS_j. It retrieves x=1, y=1.

13. Transaction k prints x=1, y=1 at the user's terminal.

Notice that it was not necessary for transaction k to use protocol P3 to obtain a correct result. It only had to wait until WRITE's from classes $\bar{i}$ and $\bar{j}$ had been processed through time TS_j, not through time TS_k (its own timestamp).

2.2.10 Protocol P2f

Protocol P2f is quite similar to protocol P2. It is used in cycles that contain a w-r-e-r-w subpath such as the subpath ($w^{\bar{i}}_{alpha}$, $r^{\bar{j}}_{alpha}$, $e^{\bar{j}}$, $r^{\bar{j}}_{beta}$, $w^{\bar{k}}_{beta}$) shown in Figure 2.11. The "f" in P2f refers to the fact that reading is being done from a <u>foreign</u> DM. As in a P2 subpath, a transaction in class $\bar{j}$ is able to observe an ordering of transactions in classes $\bar{i}$ and $\bar{k}$; protocol P2f is designed to ensure that the observed ordering is always the timestamp ordering of the transactions. If the above subpath is part of a cycle, then each transaction, j, in class $\bar{j}$ must run P2f against $\bar{i}$ at DM_{alpha} and $\bar{k}$ at DM_{beta}. This means that there must be a timestamp, say TS_o, such that all WRITE messages from $\bar{i}$ timestamped before TS_o and none timestamped after TS_o are processed before R^j_{alpha} at DM_{alpha}, and all WRITE messages from $\bar{k}$ timestamped before TS_o and none timestamped after TS_o are processed before R^j_{beta} at DM_{beta}. Protocol P2f essentially runs half of P2 (against $\bar{i}$) at one DM and half of P2 (against $\bar{k}$) at another DM.

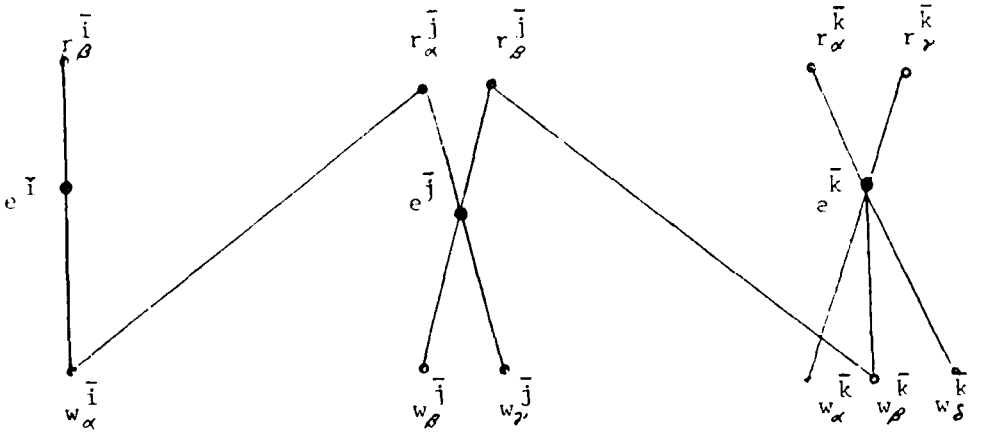


Figure 2.11 - A w-r-e-r-w subpath calls for the use of protocol P2-F when it forms part of a cycle

Since reading is being done from two separate DM's, it is not possible to use the timestamp marker mechanism. (If timestamp markers were used, it would be necessary for the two DM's involved to carry on a conversation to determine a mutually satisfactory timestamp to substitute for the marker. This kind of synchronization overhead is exactly what we are trying to avoid.) Instead, the TM issuing the READ messages chooses a timestamp (i.e., TS_o above) and includes a read condition on each READ with this timestamp. That is, if $\bar{j}$ must run P2f against $\bar{i}$ at DM_{alpha} and $\bar{k}$ at DM_{beta}, then a transaction j in class $\bar{j}$ includes the read condition $<TS_o, \bar{i}>$ in R^j_{alpha} and $<TS_o,$

$\vec{k}$> in R^j_{beta} for some chosen value of TS_o. Unfortunately, choosing a TS_o for P2f is not quite as nice as using timestamp markers in P2, because the P2f READ messages have a greater likelihood of being rejected or having to wait. The primary difference between read conditions issued as part of protocol P3 and those issued as part of protocol P2f is that the read condition timestamp for protocol P3 must be the same as the timestamp of the issuing transaction while the read condition timestamp for protocol P2f may have any value.

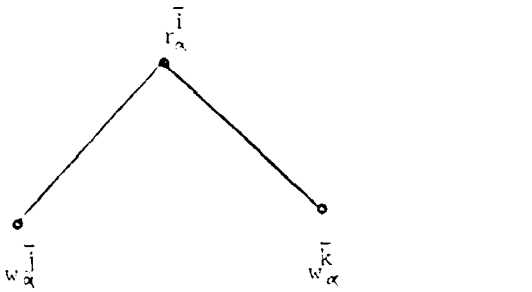
2.2.11 Protocol P1

If a transaction class appears in the graph but does not run one of protocols P2, P2f, or P3, then we say it runs protocol P1. That is to say, protocol P1 is the protocol that involves no synchronization other than the data item timestamping rule and the class pipelining rule.

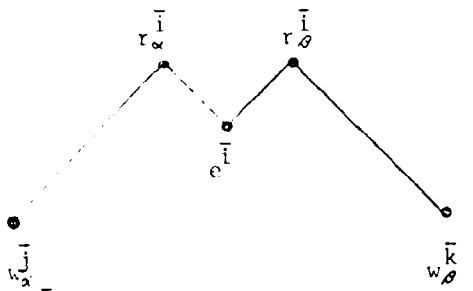
P1, P2, P2f, and P3 provide a graduated set of mechanisms in terms of concurrency and synchronization expense. A goal in designing a particular application is to distribute the data and define the classes to use the lower numbered protocols most frequently.

Graph Topology
(the subpath shown is part of a cycle)

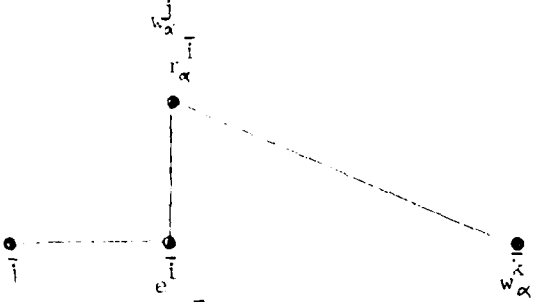
Protocol Requirement



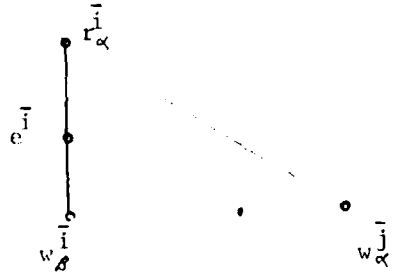
Transactions in class $\bar{i}$ must run protocol P2 with respect to classes $\bar{i}$ and $\bar{k}$.



Transactions in class $\bar{i}$ must run protocol P2-F with respect to classes $\bar{j}$ and $\bar{k}$.



Transactions in class $\bar{i}$ must run protocol P3 with respect to class $\bar{k}$.



Transactions in class $\bar{i}$ must run protocol P3 with respect to class $\bar{j}$.

Figure 2.12 - Protocol requirements are suggested by the graph topology

2.2.12 Pre-Analysis of the Class Conflict Graph

Figure 2.12 summarizes the results so far, illustrating how particular graph topologies indicate that particular protocols must be run.

If it were necessary to compute graph edges and cycles before executing each transaction, the cost of doing so would clearly be prohibitive. Fortunately, this is not necessary. The class definitions are specified by a DBA at application design time and at that time the class conflict graph can be computed and analyzed. The result of such an analysis will be a list of read conditions for each class. Note that a class may have more than one or two read conditions which it must use. This is because the class may be a part of several cycles.

When a transaction is entered at a TM, the TM first determines its read-set and write-set. It then determines to which class that transaction belongs (if the transaction can run in more than one class, the class with the fewest synchronization requirements is chosen). Having identified the transaction's class, only a table lookup is required to determine what read conditions the transaction must use.

2.2.13 Safe Cycles

It happens that there are graph cycles which never cause a non-serializable interleaving of transactions. In particular, any cycle which does not contain a vertical edge is always safe. Thus, a cycle composed entirely of diagonal edges or entirely of horizontal edges will never lead to a serializability problem and classes lying on such cycles can safely run P1 (at least insofar as the safe cycles are concerned). The cycle shown in Figure

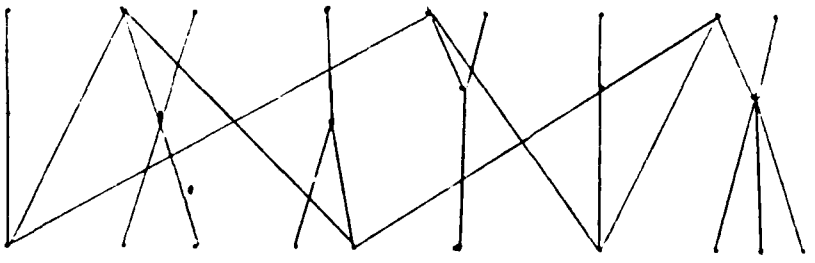


Figure 2.13 - A cycle is safe if it contains no vertical edges

2.13 is an example of a safe cycle.

This result is not immediately apparent through intuitive understanding and is illustrative of the fact that a more formal and precise treatment of serializability criteria is needed.

(Some intuitive understanding can be gained, however, through the following arguments. First, if the cycle consists entirely of horizontal edges then a serializability problem cannot arise because horizontal edges always imply a timestamp ordering of the transactions. Second, if the cycle consists entirely of diagonal edges then all the nodes on the cycle have the same DM subscript. Also such a cycle consists of a series of W-R-W subpaths. Remember from the discussion of protocol P2 that on such a subpath the reading transaction may observe a particular ordering of the writing transactions and that the observed ordering depends on the actual order in which the WRITEs were processed by the DM. Since all of the WRITEs on the cycle are being processed by the same DM, it must be the case that the reading transactions all observe the same relative ordering among the writing transactions and hence all transactions on the cycle will be serializable.)

2.2.14 Summary and Conclusions

In reviewing the concepts presented in section 2, it is helpful to distinguish between three kinds of properties of an SDD-1 system:

1. properties that are intrinsic to the way the SDD-1 software operates;

2. properties that arise from database design decisions.

3. properties that arise from the analysis of the database design.

In category (1) are the way data modules process READ messages and WRITE messages, the way clocks operate, the pipelining rules, and the way each protocol works. In category (2) are the choice of the location of SDD-1 sites on the network, the choice of logical fragments, the location of physical fragments, the configuration of materializations, the choice of read-sets and write-sets for each class, and the assignment of materializations to

each class. Finally, in category (3) is the assignment of protocols to each class.

The description we have presented of the SDD-1 redundant update mechanism has a serious defect. We have shown that certain situations cause serializability problems and have introduced mechanisms to resolve those problems. Yet how can we be sure that we have identified all possible dangerous situations? And how can we be sure that the protocols prevent all possible instances of these dangerous situations?

We believe that in order to fully understand these results, to be confident of their correctness, and to use them intelligently in designing systems, we must prove their correctness in a precise and formal manner. This has been done in [BERNSTEIN et al b].

2.3 Distributed Query Processing

In this section, the SDD-1 algorithm for processing multi-site queries is described. The primary objective of this algorithm is to execute the distributed query with the least possible delay. The algorithm assumes that the bottleneck in processing a query is the network delay. Therefore, it attempts to execute the query in such a way as to minimize the amount of data moved from one site another.

The algorithms developed here are based on several additional assumptions:

- The data model used is a relational model.
- The unit of distribution in SDD-1 is a fragment of a relation. A fragment is defined by first restricting the relation using simple booleans and then projecting those fragments on disjoint subsets of the domains to produce the final fragments. The projected fragments all share one field called the tuple id which is a unique identifier for each tuple. This facilitates tuple reconstruction by providing direct links among the sub-tuples

comprising the complete tuple. The concept of tuple id has been introduced for related purposes by [ASTRAHAN et al] and [STONEBRAKER et al].

- Although databases in SDD-1 are permitted to be arbitrarily redundant, a particular query only deals with a single non-redundant mapping of physical fragments to a complete relation.
- Only relations or logical fragments of relations are moved from one site to another.
- It does not matter which database site produces the final result.
- To facilitate cost estimation, some statistical information concerning each stored fragment is maintained. This information includes: number of tuples, which domains are inverted, the selectivity of each domain and for numeric domains the minimum and maximum values in use.

2.3.1 Achieving a Local Transaction

A relational transaction in SDD-1 is composed of a series of interdependent relational queries. In general, a relational transaction can reference data distributed over a number of database sites. The problem is to transform this distributed transaction into a local transaction using a combination of local processing and data movement. Furthermore, the transformation is required to be efficient. We shall call the process that achieves this transformation distribution.

Distribution is accomplished using two basic tactics:

a. Local processing - A fragment referenced in the transaction is reduced in size by an operation involving data at a single site. This operation is usually some combination of restriction and projection. This is usually done in anticipation of moving the reduced fragment.

b. Move Data - A fragment or a reduced fragment is moved from site A to site B. This is done with the ultimate goal of getting all the required data to one site. The algorithm does not consider the

possibility of moving a join of two relations even if this were to prove advantageous.

The objective in optimizing distribution is to minimize the combined cost (in terms of time) of local processing and data movement. Since it has been assumed that data movement is slow compared to local processing, minimization of data movement is the primary focus of the optimization.

2.3.1.1 The Approach

The simplest approach to executing a distributed query is to pick one site, move all the data there and execute the query. This approach obviously works, but is has two drawbacks. First, the amount of data moved from one site to another might be very large and therefore the move might be very time consuming. Second, this approach does not exploit the potential benefits of distributed processing occurring in parallel.

The first improvement that can be made to this method is to choose the site that has the largest relation as the final site so that this relation does not have to be moved. However, if there are two or more sites with large

amounts of data, this won't help much. What is needed is a means of reducing the size of a relation prior to moving it. This leads to the next improvement to the algorithm.

Even though the distributed query deals with relations at many sites, often there are subqueries that only deal with data at one site. In many cases, processing the local subqueries reduces the size of the moved relation significantly.

Consider the following example:

relation	site	domains
R_1	A	ALPHA, BETA
R_2	B	ALPHA, GAMMA

```
RANGE OF X IS R1
RANGE OF Y IS R2
RETRIEVE X.ALPHA
  WHERE (X.BETA=1)AND(Y.GAMMA=2) AND
        (X.ALPHA = Y.ALPHA)
```

In the above example, it is clear that X.BETA = 1 is a local subquery and Y.GAMMA = 2 is also. It is also obvious that executing these local subqueries first will reduce the sizes of R_1 and R_2 and regardless of the final site will reduce the amount of data moved.

The next improvement to the algorithm is the least intuitive. Basically, the optimization involves moving data to a non-final site where it can be used to restrict the data at that site such that less data is moved to the final site. In order for the optimization strategy to work, the combined cost of the first move, processing and second move must constitute an overall improvement over the original moves. This part of the algorithm is obviously recursive in that it can be re-applied to the new set of moves to achieve further improvement.

The algorithm thus far presented has some obvious shortcomings. It suffers from the usual hill-climbing problem in that it may find a local optimum. Because it attempts to find only one final site, it also fails in the following examples:

Example 1:

Relation	Site	Domains
R_1	A	X,Y
R_2	B	X,Y

RETRIEVE R_1.X

WHERE (R_1.X=R_2.X) AND (R_1.Y=1 OR R_2.Y=1)

The optimal strategy would be to compute the restrictions $R_1.Y=1$ and $R_2.X=1$ at their local sites. Then move the restricted R_1 to B and the restricted R_2 to A. Then $R_1.X=R_2.X$ can be done locally at both A and B and finally the results would be unioned at either site. In order to come up with this strategy, the algorithm must include the concept of multi-final sites.

Example 2:

Using the same relations

RETRIEVE ($R_1.X$, $R_2.X$)

WHERE ($R_1.X = 1$ AND $R_1.y = 2$) OR

($R_2.X = 1$ AND $R_2.y = 2$)

The optimal strategy in this case would be to do each half of the OR locally and union the result at either site. This also requires the concept of multi-final sites.

The basic algorithm has additional improvements to deal with some of these problems. The first such improvement is designed to reject poor local optimums. The reason poor local optimums can be chosen is that the final site is chosen based on least upper-bound analysis. This is because the final site choice assumes no improvement in the initial move set. The algorithm is improved by including a branch and bound technique to examine multiple

possible final sites. So far, final site selection has been based on an upper-bound analysis since the selection process assumes no improvement in the initial move set. In the augmented algorithm, each site is also examined assuming the best possible case of move optimization. The cost of the moves in this case represents a lower-bound on the cost of choosing each site as a final site. The lower-bound cost is computed by assuming all data that is not at this site initially is moved to one other site and all possible local processing takes place there. The lower bound on the site's cost is the cost of getting the reduced data to the site under consideration. Any site whose lower-bound cost is greater than some other site's upper-bound cost is rejected as a possible final site. The remaining sites can now all be considered as potential final sites. If the number of potential final sites is too large, each potential final site gets the average of its upper and lower-bound costs computed and only the lowest N are kept. N is a number that can be determined experimentally to give reasonable performance. Further optimization will occur to each set of moves associated with each potential final site. As the optimization steps occur, upper-bounds will get lower so that more sites may be rejected.

Consider the following example:

Example 3:

Relation	Site	Domains	Size of Relation
R_1	A	X,Y	1000
R_2	B	X,Z	2000
R_3	C	Z,W	1000

RETRIEVE R_1.X

WHERE R_1.Y = 1 R_1.X = R_2.X R.Z = R_3.Z

Assuming that R_1:R_1.Y=1 is 100 tuples, the original algorithm would choose site B for the final result. However, the lower bound analysis would show site C as having the best lower-bound and in fact the optimal solution is achieved with C as the final site.

The other two problematic cases are solved by loosening the requirement that all moves eventually result in all data ending up at a single final site. When dealing with queries that include non-local disjunctions, the algorithm converts the restriction to disjunctive normal form and treats each disjunction as a separate query. After creating results at various sites for each part of the disjunction, it unions the results at one particular site.

The remainder of this paper details the move planning algorithm including the relational query to fragment query

transformation algorithm and the cost estimation algorithm.

2.3.1.2 Movement Strategy

The overall optimization strategy used in SDD-1 consists of the following steps:

a. Do all the local processing that can be done without moving any data.
b. For each site, compute the upper and lower-bounds on the cost of choosing that site as the final site. Reject any sites whose lower-bound is greater than some other site's upper-bound. If there are more than N sites left, compute the average of the upper and lower-bound costs and keep the N sites with the lowest average. For each site left, compute the set of moves to get all the data to that site and call that set of moves M_0 for that site.
c. For each M_0, try to find two successive sets of moves M_1 and M_2 with additional local processing between them that produce the same result as M_0 but have a combined cost including the additional local processing less than that of M_0. If no such set

exists, $M_\emptyset$ is optimal. If more than one such set exists, choose the least expensive. If $M_\emptyset$ is replaced, update the upper-bound for the site and attempt to eliminate other sites whose lower bound is now greater than this new upper-bound.

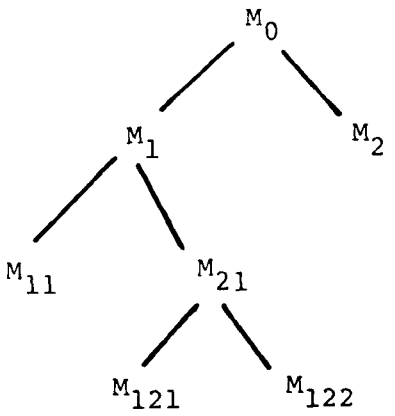
d. Iterate (c) on M1 and M2.

The successive iterations of (c) in the above algorithm can be represented as a binary tree, where $M_\emptyset$ is the root, each leaf represents a set of moves, the leaves are executed in sequence from left to right and local

---

Example

Figure 2.14



---

processing takes place between successive leaves. The moves take place as follows:

$M_{11} - M_{121} - M_{122} - M_2$

The remainder of this report spells out in detail the steps in the strategy of moves.

2.3.2 Query in Terms of Fragments

The first step in processing a distributed transaction cannot be taken until it has been transformed from a transaction in terms of relations to a transaction in terms of fragments. This is necessary to determine the whereabouts of the various data fragments to be used in the transaction.

2.3.2.1 Fragment Definition

As mentioned previously, a fragment is formed by first restricting and then projecting the tuples in a relation. Specifically, let R be a relation with domains D_0, D_1, D_2 ... D_n. D_0 is the tuple-id domain. The general subdivision of R into fragments is done as follows:

a. First R is partitioned horizontally into sets of tuples based on simple booleans. The different horizontal partitions may be expressed in terms of $D_\emptyset$:

   $D_\emptyset j \quad j=1,2,3 \ldots m$

   i.e. $D_\emptyset j$ represents the set of $D_\emptyset$s for the tuples in horizontal partition j.

b. Each horizontal partition may then be partitioned vertically. For each horizontal partition j, disjoint sets $S_1^{(j)}$, $S_2^{(j)}$, ... $S_k^{(j)} j$ of domains D_1, D_2, ... D_n are defined. The domain $D_\emptyset$ plus the domains in S_i^j constitute the vertical partitions.

c. The fragments are obtained by subdividing the tuples according to $D_{\emptyset j}$ and projecting the resulting partitions on $\{D_\emptyset, S_i^j\}$. I.e.

   $F_i^{(j)} = \{r[D_\emptyset S_i^j] : r \in R \text{ and } r[D_\emptyset] \in D_{\emptyset j}\}$

2.3.2.2 General Form of a Query

Assuming the relational query being transformed is equivalent to a QUEL [HELD et al] statement, references to a relation R will be through a tuple variable, say X. In general a query may contain several tuple variables each ranging over a different relation. The transformation procedure will be to change each variable one-at-a-time into a set of variables that reference the fragments. Focusing on a single variable X, a general query is of the form:

RANGE OF X IS R

RETRIEVE TL(X) where Q(X)

TL is the target list and Q the qualification. The goal is to express the query as one or more queries of the form:

RANGE OF $x_i(j)$ IS $F_i^{(j)}$ $j = 1,2,3...m$ $i=1,...k$;

RETRIEVE TL ($\{X_i^{(j)}\}$) WHERE Q($\{X_i^{(j)}\}$)

2.3.2.3 Transformation Algorithm

The overall approach will be to consider the horizontal fragmentation (division of the tuples) first and vertical fragmentation (division of domains) second. Therefore X will first be replaced by X_j $j=1,2...m$ and then each X_j will be replaced by $X_j^{(i)}$ $i=1,2,...k;$.

2.3.2.3.1 Horizontal Fragmentation

a. Range Declaration

RANGE OF X IS R

becomes a series of Range declarations of the form:

RANGE OF $X_i^{(j)}$ IS $F_i^{(j)}$ $j=1,2,3,...m$ $i=1,...k;$

WHERE $F_j = {}_i F_j^{(i)}$

b. Retrieve Statement

The retrieve statement:

RETRIEVE TL(X) WHERE Q(X)

by a sequence of queries

RETRIEVE TL(X_j) WHERE Q(X_j) $j=1,2,3,...m$

and take the union of the results.

Since horizontal fragmentation will be defined by some predicate on the domain, the defining predicate for a fragment represents an integrity constraint on that fragment. For each horizontal fragment, F_j, its defining predicate, P_j, must be ANDed in to the qualification $Q(X_j)$. In some cases this additional predicate may conflict with $Q(X_j)$. If this occurs, no access to that fragment need occur. For example, if the horizontal fragments are defined as:

F_1 = {r : r[Salary] $\leq$ 20K}

F_2 = {r : r[Salary] > 20K}

and a query says:

RANGE OF r IS R

RETRIEVE r[name] WHERE r[Salary] < 15K

then only F_1 need be examined since Salary can never be both > 20K and < 15K at the same time.

2.3.2.3.2 Vertical Fragmentation

After the transformation for horizontal fragmentation, each query from the sequence of queries produced can be treated as a normal query. Therefore, vertical fragmentation can be dealt with as if horizontal fragmentation didn't exist with no lack of generality.

a. The range declaration:

RANGE OF X IS R

becomes:

RANGE OF $X^{(i)}$ IS $F^{(i)}$

i=1,2,3,...k

b. Target List

For each term X.D determine the S_i to which D belongs. Then replace X.D by $X^{(i)}$.D.

c. Qualification

i) Replace X.D by $X^{(i)}$.D if D S_i.

ii) For those $X^{(i)}$ which appear in the target list or qualification, an additional qualification must be added to force the sub-tuples to link together correctly. The additional qualification required is "mutual equality on $D_\emptyset$". For example, if $X^{(1)}$, $X^{(2)}$, and $X^{(3)}$ appear, the condition ($X^{(1)}.D_\emptyset = X^{(2)}.D_\emptyset$) and ($X^{(1)}.D_\emptyset = X^{(3)}.D_\emptyset$) must be added.

2.3.2.4 Summary of Query Transformation

a. The transformation is undertaken one variable at a time, even when several tuple-variables reference the same relation.

b. The transformation is performed first for horizontal fragmentation and then for vertical transformation.

c. The transformation rule for horizontal fragmentation is to repeat the query for each horizontal fragment and then take the union of the results. In addition, the fragment defining boolean is ANDed into the qualification for each fragment.

d. The transformation rule for vertical fragmentation is to replace a variable Which references a relation by one that references a fragment according to which domain is being referenced and to add to the qualification clauses that link the key-domains of the fragments.

2.3.2.5 An Example

Consider the following database:

R D_0 D_1 D_2 D_3 D_4

Employee(E#,Name,Salary,Dept,Mgr)

This relation has the following fragment definitions associated with it:

a. Horizontal Fragmentation

F_1 = Employee [E.Salary < 20 K]

F_2 = Employee [E.Salary > 20 K]

b. Vertical Fragmentation

For F_1: $F_1^{(1)} = F_1$ [E#,Name,Salary,Mgr]

$F_1^{(2)} = F_1$ [E#, Dept]

For F_2: $F_2^{(1)} = F_1$ [E#,Name,Salary,Dept]

$F_2^{(2)} = F_2$ [E#, Mgr]

Query: "Find names of all employees who earn less than 15K".

```
RANGE OF X IS R
RETRIEVE (E.Name) WHERE E.Salary < 15K
```

a. Transform for Horizontal Fragmentation

RANGE OF X_1 IS F_1

RETRIEVE $T_1(X_1.\text{Name})$ WHERE $(X_1.\text{Salary} < 15K)$ AND $(X_1.\text{Salary} \leq 20K)$

RANGE OF X_2 IS F_2

RETRIEVE $T_2(X_1.\text{Name})$ WHERE $(X_1.\text{Salary} < 15K)$ AND $(X_1.\text{Salary} > 20K)$

In theory, the obvious boolean reductions are possible:

$(a < 15K)$ and $(a \leq 20K) = (a < 15K)$

$(a < 15K)$ and $(a > 20K) =$ False

but to do so in practice may be non-trivial. Assuming this can be done, the new query becomes:

RANGE OF X_1 IS F_1

RETRIEVE $T_1(X_1.\text{Name})$ WHERE $X_1.\text{Salary} < 15K$

b. Vertical Fragmentation

RANGE OF $X_1^{(1)}$ IS $F_1^{(1)}$
RANGE OF $X_1^{(2)}$ IS $F_1^{(2)}$
RETRIEVE $T_1(X_1^{(1)}.\text{Name})$ WHERE $X_1^{(1)}.\text{Salary} < 15K$

Since $X_1^{(2)}$ does not appear the linking term $E\#.X_1^{(1)} = E\#.X_1^{(2)}$ is not required. Also the Range for $X_1^{(2)}$ may be dropped out.

2.3.3 Initial Local Processing

Once the distributed transaction has been redefined in terms of the logical fragments, the global directory may be consulted to find the locations of the physical fragments. In order to keep the notation simple, the rest of this discussion will assume the unit of distribution is an entire relation instead of a fragment. This assumption loses no generality since it has already been shown how to do the appropriate transformation.

A relational transaction T consists of a sequence $T_1, T_2, \ldots T_K$ where each T_i is a set of relational queries that can be processed in parallel and T_i makes use of the results from $T_1, T_2 \ldots T_{i-1}$. It is assumed that every query is expressed in terms of tuple-variables and each tuple-variable references either a relation stored at a single site or a result relation of an earlier query.

Consider the following example:

Location	Relation
A	EMPLOYEE (ENAME, D#, SALARY)
B	DEPT (D#, MANAGER, #EMP)

T_1: RANGE OF D IS DEPT
RETRIEVE EXECS (D.D#,D.MANAGER)
WHERE (D.#EMP > 50)

T_2: RANGE OF E IS EMPLOYEE
RANGE OF X IS EXECS
RETRIEVE WELLPAID (E.ENAME)
(E.D# = X.D#) AND (E.SALARY > 50K)

The example deals with two relations stored at two data modules. EMPLOYEE contains the employee's name, department number and salary and it is stored at site A. DEPT contains the department number, its manager and the number of employees and it is stored at site B. The query asks for all employees who make more than 50K and work in departments with more than 50 employees.

The first step in processing the query is to change the tuple variables in the transaction to include the location information. This is done by concatenating a location variable L(V) to each variable V. L(V) can take on the value of any site or the value U if the location is currently unknown. Initially, if V references an existing relation in the database then L(V) takes the value of the site where relation is located. If V references a result relation then L(V) = U since its location has yet to be determined. As processing of the transaction proceeds and data is moved, L(V) changes for the affected variables.

In transaction T of the example, the initial values of location variables are:

Tuple variable	Location variable
D	B
E	A
X	U

and the retrieve statement becomes:

T_1: RETRIEVE EXECS (B.D.D3, B.D.MANAGER)
WHERE (B.D.#EMP > 50)

T_2: RETRIEVE WELLPAID (A.E.ENAME)
WHERE A.E.D# = U.X.D#) AND (A.E.SALARY > 50K)

The process of determining what local operations can be made before data must be moved is now straightforward. Each query in T is examined and any subquery that references a single known location is extracted. The results of any such local operation is a relation at that site and will give some of the tuple variables with unknown locations definite location values. A second round of local processing may now be possible and so forth. The procedure is continued until no more sub-queries can be extracted without moving data.

For the previous example, the following one-location sub-queries can be extracted:

From T_1 T_1 itself

From T_2 RETRIEVE EMP´ (A.E.ENAME,A.E.D#)
WHERE (A.E.SALARY > 50K) Q_2

After T_1 is executed, the location variable of X in T_2 becomes B. After the execution of both T_1 and Q_2 what remains of the transaction T is simply

```
       RANGE OF (E,X) IS (EMP',EXECS)
   T'  RETRIEVE WELLPAID (A.E.ENAME)
          WHERE (A.E.D# = B.X.D#)
```

It is obvious that further local processing is impossible without moving data.

The procedure used to determine what local operation can be performed consists of repeated application of the following two steps:

a. Extract one-location subqueries
b. Assign location values for the result relations and repeat (a).

The algorithm for carrying out (a) is exactly the same as that for one-variable subquery extraction in decomposition [WONG and YOUSSEFI]. Basically, it involves expressing the query in conjunctive form and detaching clauses which involve only one location.

One question discussed in [WONG], but not pursued here is whether local processing which is not explicit in the transaction but implied by transitivity should be carried out. For example,

(a > b) and (b > 0) imply (a > 0)

In this case, a > 0 could represent additional useful local processing.

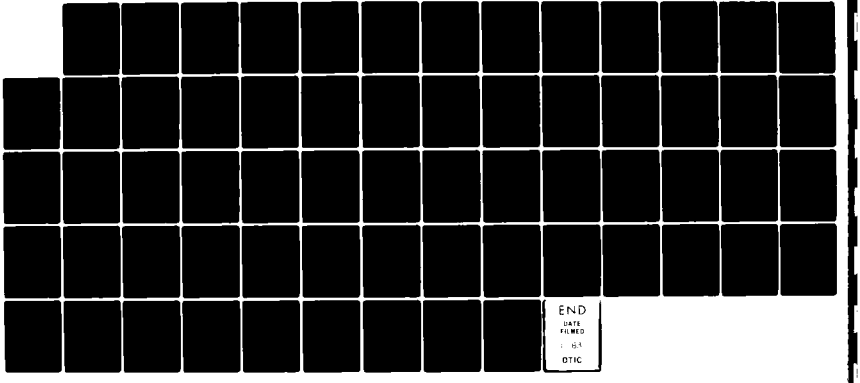
The decomposition algorithm is modified slightly to handle disjunctions as mentioned previously. This change involves expressing the query in disjunctive normal form and then treating each disjunction as a separate query and finally unioning the results. The algorithm is careful not to perform duplicate local processing that may appear as a result of the transformation to disjunctive normal form.

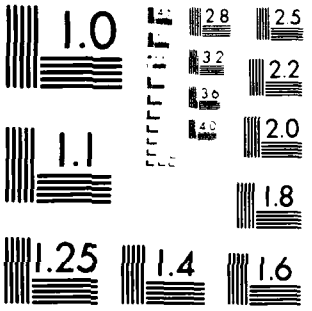
2.3.4 The Initial Move Sets M_0s

After all possible local processing has occurred, it is necessary to choose the initial sets of moves M_0s. For each site, M_0 is the set of moves to get all the data to that site. In other words, M_0 is a set of moves all of whose destinations are the same site and this site is the site where the final result would be produced. The cost of performing the moves in M_0 represents the upper-bound cost of using that site as the final site. In addition, for each site compute the lower-bound cost of that site. This can be done by assuming that all data not already at

AD-A122 416 A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTROL APPLICATIONS(U) PRESRAY CORP PALO ALTO CALIF 30 JAN 78 CCA-78-03 N00039-77-C-0074 2/2

UNCLASSIFIED F/G 9/2 NL





that site is at one other site and all possible local processing has occurred at that site. The lower-bound is the cost of getting the resulting data to the potential final site. Any sites with lower-bounds greater than some other site's upper bound is rejected as a potential final site and only the best N of the remaining sites are kept based on the average of the upper and lower-bounds.

2.3.4.1 Cost of Data Movement

In order to compare the costs of various potential $M_{\emptyset}$'s, there must be some way of estimating the time required by each set of moves. In the following, M is a set of moves.

In computing cost ($M_{\emptyset}$), two assumptions are made:

1. The cost of moving data is directly proportional to the amount of data moved.
2. Data movement between different source-destination pairs can occur in parallel.

For a given M, let S(M) be the set of distinct datamodule source-destination pairs in M. For each pair p in S(M), let Dp be the amount of data to be transferred between that pair and let Bp be the effective bandwidth of the

Communications Channel between that pair of sites. Based on the parallel assumption,

$$\text{Cost}(M) = \max_{p \in S(M)} \left(\frac{Dp}{Bp} \right)$$

In order to compute Dp, the following information must be known:

a. The fragments that must move between the pair p. These will be called $R_1(P)$, $R_2(P)$, $R_3(P)$ ... $R_K(P)$.

b. The number of tuples in $R_i(P)$; this is denoted as $|R_i(P)|$

c. The width, W_i (in units of Dp) of each tuple in $R_i(p)$

In terms of these quantities:

$$Dp = \sum_i W_i \quad R_i(P)$$

and

$$\text{Cost}(M) = \text{Max}_{P \in S(M)} \frac{1}{Bp} \sum_i W_i \left| R_i(P) \right|$$

All that is needed is a way of computing $|R_i(P)|$. This will be discussed in the following section.

2.3.4.2 Estimating Local Processing Costs and Result Sizes

The three basic operations involved in local processing during a distributed query are: restriction, projection and join-project. Join-project is considered the basic operation instead of just join, since joins are never moved unless they are first projected on one of the joined relations.

2.3.4.2.1 Cost of a Restriction

A restriction is the selection of those tuples in a relation that satisfy a given condition. If R is a relation and C is a condition such that every r in R either satisfies C (C(r)=true) or fails to satisfy C (C(r)=false), the restriction of R by C may be denoted as:

$$R[C] = \{r : r \in R \text{ and } C(r)=\text{true}\}$$

The quantities of interest are q_C, the number of tuples that satisfy C and m_C, the number of tuples to be examined that satisfy C. To keep the cost estimation simple, it is

assumed that the domain values are independent and evenly distributed. This assumption trades-off accuracy against simplicity of computation.

In order to make the estimates, the concept of selectivity is defined for domain d.

If d is non-numeric:

$$S_{(d=k)} = \frac{1}{\text{number of unique d values}}$$

If d is numeric:

$$S_{(d=k)} = \frac{1}{\text{max-min}}$$

$$S_{(d>k)} = \frac{\text{max} - k}{\text{max-min}}$$

$$S_{(d<k)} = \frac{k - \text{min}}{\text{max-min}}$$

In other words the selectivity specifies the fraction of records that satisfy a boolean. The following extensions of selectivity are obvious assuming independence:

$$S_{A \wedge B} = S_A * S_B$$

$$S_{\neg A} = (1 - S_A)$$

$$S_{A \vee B} = S_A + S_B - S_A * S_B$$

The number of records selected by a restriction is:

$q_c = S_c \ |R|$ where $|R|$ is the number of records in R.

The number of records to be examined depends on which fields are inverted. In a restriction based on a simple boolean of the form:

$$\text{domain} \left(\begin{matrix} = \\ \neq \end{matrix}\right) \text{ constant}$$

If the field is inverted, exactly $S_{(d=c)} \ |R|$ records are examined. If the field is not inverted or the operator is not = or ≠, then all records must be examined. The following obvious extensions exist once again:

$A_{INV} \wedge B_{INV} \quad M_c = S_A * S_B * |R|$

$A_{INV} \wedge B_{NOT} \quad M_c = S_A * |R|$

$A_{NOT} \wedge B_{INV} \quad M_c = S_B * |R|$

$A_{NOT} \wedge B_{NOT} \quad M_c = |R|$

$\neg A_{INV} \quad M_c = (1 - S_A) \ |R|$

$\neg A_{NOT} \quad M_c = |R|$

$A_{INV} \vee B_{INV} \quad M_c = (S_A + S_B - S_A * S_B)|R|$

$A_{INV} \vee B_{NOT} \quad M_c = |R|$

$A_{NOT} \vee B_{INV} \quad M_c = |R|$

$A_{NOT} \vee B_{NOT} \quad M_c = |R|$

Therefore, from selectivities and inversion information, both the number of records returned and the number of records examined may be determined.

2.3.4.2.2 Cost of a Projection

Assume a relation R with domains D_1, D_2, D_3 ... D_n. The cost of projecting R on some subset of the domains Di_1, Di_2, Di_3...Di_m must be determined. The number of tuples in the projection can be no greater than the number of tuples in R. It is also bounded by the product of the numbers of possible values for each projected domain. Therefore:

$$|R[Di_1, Di_2 \ldots Di_m]| = \min(|R|, \prod_{j=1}^{m} |Di_j|)$$

where |Di| is the number of unique values in domain i. This result can be improved since it is not generally that large :

$$|R[Di_1, Di_2 \ldots Di_m]| = \min(|R|, C_m \prod_{j=1}^{m} |Di_j|)$$

where C_m is a decreasing function of m (C_1 = 1) which should be empirically determined.

The number of tuples that must be touched to compute the projection is more difficult to estimate. The main problem is duplicate elimination. If one of the projected domains is a key, then no duplicate elimination is required and there will be exactly |R| tuples in the result and exactly |R| tuples need be examined. Otherwise, something must be done to eliminate duplicates. Assuming that either inversions or some kind of hashing is used, approximately 2 * |R| tuples must be examined.

2.3.4.2.3 Cost of a Join-Project

If R and S are relations, then:

$$R[(D_1 = D_2)S] = (r,s):\ r \in R,\ s \in S, \text{and } \left.\begin{array}{l} \\ r \cdot D_1 = s \cdot D_2 \end{array}\right\} \text{projected on } R$$

This can be calculated in either of two ways:

a. For each tuple in R scan S for a matching domain and project.

b. For each tuple in S scan R for a matching domain and project.

In either case duplicate elimination is required. Let $q_r(S)$ denote the number of S-tuples that satisfy the condition for a giver r and let $m_r(S)$ denote the number of S-tuples that must be examined to determine those that qualify. Similarly, $q_s(R)$ and $m_s(R)$ can be defined. Since the join-project can be done in either of the two ways mentioned and if it is assumed (optimistically) that duplicates can be eliminated by incurring a cost approximately equal to the number of tuples in the join, then:

$$\text{cost(join-project)} = \min \{ \sum_{r \in R} [M_r(S) + q_r(S)], \sum_{s \in S} [M_s(R) + q_s(R)] \}$$

In each sum, the first term represents the cost of the join and the second that of eliminating duplicates.

In practice, it would be more reasonable to replace q_r, m_r, q_s and q_r by estimates of their average values. These may be represented as:

$$m(s) = \text{Avg } M_r(S)$$

etc.

If this is done, then:

$$\text{cost(join-project)} = \min \{ |R| (m(S) + q(S)), |S| (m(R) + q(R)) \}$$

Since the number of tuples in the join must be the same regardless of the method of computation.

$$|R|q(S) = |S|q(R)$$

The relationship between m and q is the same as that described in 2.3.4.2.1, the only required quantities to be computed are $|R|$, $|S|$, and either $q(S)$ or $q(R)$. $|R|$ and $|S|$ are either given as part of the available information or calculated from previous operations. What remains is to calculate $q(R)$ or $q(S)$. This can be done by simply calculating the size of the join-project.

In the join-project:

$$(R(V_1 = V_2)S) \text{ [domains of R]}$$

there are two extreme cases to consider:

a.

$R[D_1] \subset S[D_2]$ or visa versa

in which case

$$|R(D_1 = D_2)S \text{ [domains of R]}| = \min \frac{(|R[D_1]|, |S[D_2]|)}{|R[D_1]|}|R|$$

b.

$R[D_1]$ and $S[D_2]$ are "independent"

in which case

$$|(R(D_1 = D_2)S) \text{ [domains of R]}| = \frac{|S[D_2]|}{|D_2|}|R|$$

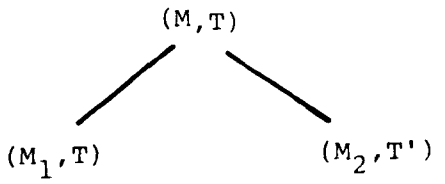
where $|D|$ is the number of possible values for $|D|$.

2.3.5 Optimizing the Initial Move Set

Returning to the discussion of distributed query optimization, the next order of business is to optimize the initial move set. As mentioned in section 2.2 - the optimization algorithm involves building a binary tree of moves by successively splitting the nodes in the move tree. In order to narrow the range of possibilities, a node M is only split into two nodes M_1 and M_2 if the combined cost of M_1, M_2 and required local processing are less than the cost of M.

In order to make the process of node splitting recursive, each node in the tree has not only a set of proposed moves M associated with it, but also the form of the transaction before the moves in M are made. Hence, each node has a

set of moves M and a transaction T associated with it. To split (M,T) means replacing the moves in M by the moves in M_1, followed by local processing and then the moves in M_2. Graphically, this split may be represented by:



where T´ is T as modified by M_1 and resulting local processing.

2.3.5.1 Identifying the Potentially Reducing Moves

Each m in M involves moving a fragment. The requirement that M be split only if there is an immediate improvement means that the split must decrease the size of at least one fragment in M. The only way to reduce the size of a fragment, F, is to cause a subquery involving F to become local. This can be done by moving to some other fragment that is linked to F in the transaction to F´s site. In other words, if fragment F_m is associated with move m and F_m is at site A and referenced by tuple variable X, a clause of the form:

C(A . X, B . Y)

would suggest that moving the fragment associated with Y from B to A might reduce F_m. Also, if there is a clause of the form:

C(A.X, U.Y)

would suggest that a move combined with local processing that could change U to A might reduce F_m.

By examining every clause in the transaction that is associated with a fragment in the move set, an entire set of potentially reducing moves can be generated.

2.3.5.2 Determining the Actual Reducing Moves

The set of all potentially reducing moves is first examined one move at a time. Any move that reduces more than it costs is put into M_1. Then, any pair of moves that reduce more than they cost and don't include those already put into M_1 are added to M_1. And so forth for three, four, etc. M_2 is generated by performing all local processing after M_1 and putting the resulting moves plus moves in M that weren't modified into M_2. The transaction associated with M_2 is the original transaction modified by M_1 and resulting local operations. The procedure may then

be recursively applied to each branch of the tree until no moves can be put into M_1.

2.3.5.3 An Example

The following example is an illustration of how the algorithm works.

Relation (domains [Selectivity])	Site	Var in Query	TOTAL#TUPLE
Supplier(S#[.05],City[.1])	A	S	20
Availability(S#[.05],P#[.01], QOH[0-2000])	A	V	2000
Parts(P#[.01],Pname[.1])	B	P	100
Projects(J#[.01], City[.1])	C	J	100
Supply(J#[.01],S#[.05],P#[.01], QTY[0-2000]	C	Y	10000

```
RETRIEVE (S  S#)

WHERE (S.S# = V.S#) AND S. City =  J.  City)  AND  S.S#  =
      Y.S#)  AND  (V.P#  =  P.P#) AND (V.QOH > Y.QTY) AND
      (P.P# = Y.P#) AND (P.Pname = 'Bolts') AND  (J.J#  =
      Y.J#) AND (Y.QTY > 1000)
```

Stated verbally, the query asks for the S# for those suppliers who supply bolts to projects located in the same city in quantity greater than 1000 and for whom the quantity-on-hand is larger than the quantity supplied. To keep the analysis simple, the relations in this example are not fragmented.

Step 1: Add location to variables

```
RETRIEVE (A.S.S#)

WHERE (A.S.S# = A.V.S#) AND (A.S.City = C.J.City) AND
      (A.S.S# = C.Y.S#) AND (A.V.P# = B.P.P#) AND
      (A.V.QOH > C.Y.QTY) AND (B.P.P# = C.Y.P#) AND
      (B.P.Pnam = 'Bolts') AND (C.J.J# = C.Y.J#) AND
      (C.Y.QTY > 1000)
```

Step 2: Identify Initial Local Processing

The following local subqueries exist. It should be noted that because of the restriction that only subrelations, projections of relations or a combination of the two are moved, the obvious joins are not considered for movement.

```
A.S.S# = A.V.S#
B.P.Pname = 'Bolts'
C.J.J# = C.Y.J#
C.Y.Qty > 1000
```

The residual query after removing the local subqueries is:

```
RETRIEVE (A.S.S#)

WHERE (A.S.S# = A.V.S#) AND (A.S.City = C.J.City) AND
      (A.S.S# = C.Y.S#) AND (A.V.P# = B.P.P#) AND
      (A.V.QOH > C.Y.QTY) AND (B.P.P# = C.Y.P#) AND
      (C.J.J# = C.Y.J#)
```

Note that the two local joining clauses (A.S.S# = A.V.S#) and (C.J.J# = C.Y.J#) have to remain in the modified query since the join is not moved, just the projected join.

The residual query indicates which domains of the relations must be retained to complete the query.

Site	Domains to Retain
A	A.S.S#, A.V.S#, A.S.City, A.V.P#, A.V.QOH
B	B.P.P#
C	C.Y.S#, C.Y.P#, C.Y.Qty, C.J.City, C.J.J#, C.Y.J#

The definitions of the relations that actually result from the local processing are:

Site	Relation	Definition
A	Asupplier	(A.S.S#, A.S.City)
	AAvailability	(A.V.S#,A.V.P#, A.V.QOH)

B	BParts	(B.P.P#)
C	CSupply	(C.Y.S#,C.Y.P#,C.Y.J# C.Y.Qty)
	CProject	(C.J.J#,C.J.City)

Step 3: Determine M_0 The estimated sizes are:

Relation	Size(# of tuples)
ASupplier	20
AAvailability	2000
BParts	10
CSupply	5000
CProject	100

The possible final destination and the associated movement costs are:

Final Destination	Frags to Move	Cost
A	BParts CSupply CProject	5100
B	ASupplier,AAvailability CSupply,CProject	5100
C	ASupplier,AAvailability, BParts	2100

The obvious choice for a final destination is C (based on an upper bound analysis). In this particular case, a

lower bound analysis indicates C is the best final site also.

Site	Improvements	Cost
A	CSupply reduced by BParts	600
B	No Useful Reduction	5100
C	AAvailability reduced by BParts	300

Therefore, M_0 is:

$$M_0 = \begin{cases} m_1 = \{\text{Move ASupply from A to C}\} \\ m_2 = \{\text{Move AAvailability from A to C}\} \\ m_3 = \{\text{Move BParts from B to C}\} \end{cases}$$

Step 4: Find all Potentially Reducing Moves

The possible moves that might reduce M_0 are found by the following method:

For each fragment referenced by a move in M, find the subqueries involving that fragment. If another fragment in the subquery is not at the same site, moving this other fragment to the original fragments site is a potentially reducing move.

Fragment ASupplier is referenced in subqueries:

(A.S.S# = A.V.S#)

(A.S.S# = C.Y.S#)

(A.S.City = C.J.City)

This indicates two potentially reducing moves:

m_{11} = Move CSupply[S#]from C to A

M_1 m_{12} = Move CProject[City]from C to A

In a similar manner, for AAvailability:

(A.V.S# = C.Y.S#)

(A.V.P# = C.Y.P#)

(A.V.P# = B.P.P#)

(A.V.QOH > C.Y.QTY)

m_{21} = Move CSupply [S#, P#, Qty] from C to A

M_2 m_{22} = Move BParts [P#] from B to A

And for BParts:

m_{31} = Move AAvailability [P#]from A to B

M_3 m_{32} = Move CSupply [P#] from C to B

In order for a potentially reducing move to be included, it must cost less than the reduction it effects.

The costs and reductions of the proposed reducing moves are:

candidate		cost	reduction
m_{11}	CSupply [S#] (ASupplier(S#=S#)CSupply) [S#,City]	10	ASupplier reduced to no substantial reduction
m_{12}	CProject [City] (ASupplier(City=City)CProject) [S#,City]	10	ASupplier reduced to no substantial reduction
m_{21}	CSupply [S#, P#, Qty] (ASupplier(S#=S#)CSupply) and AAvailability reduced to (AAvailability (S#=S#,P#=P#,QOH>Qty) CSupply) [S#,P#,QOH]	5000	ASupplier reduced to no substantial reduction
m_{22}	BParts[P#] (AAvailability(P#=P#)BParts) [S#,P#,QOH]	10	AAvailability reduced to AAvailability reduced 2000 -> 200
m_{31}	AAvailability [P#] (BParts (P#=P#)AAvailability) [P#]	100	BParts reduced to no substantial reduction
m_{32}	CSupply[P#] (BParts(P#=P#)CSupply) [P#]		BParts reduced to no substantial reduction

From the given analysis, m_{22} is the only reducing move.

$$M_1 = \begin{matrix} m_1 = \text{Move ASupplier from A to C} \\ m_{22} = \text{Move BParts from B to A} \end{matrix}$$

Note that m_3 was not included in M_1 because BParts is not at B anymore. However, a possible alternate strategy would be to leave a copy of BParts at B. In this case, m_3 would stay in M_1.

The resulting M_2 will be:

m'_2 = Move (AAvailability (P#=P#) BParts)
[S#,P#,QOH] from A to C

m'_2 = Move BParts from A to C

m'_3 can be considered redundant since it is part of m'_2.

Thus the algorithm has yielded an optimal move strategy for this query.

2.4 Reliability

2.4.1 Overview

The problem that we address in this section is that of ensuring the continued correct operation of SDD-1,in the presence of failures (and recoveries) of individual sites of the system and of failures of the communications system connecting them. This problem is analogous to the issues of reliability, integrity, and recovery in a single-site data base system, but the two problem areas are fundamentally different. In a single-site environment, the emphasis is on maintaining at all times sufficient information, in a form that can survive the failure of the system, to allow the reconstruction of an accurate and consistent version of the data base when the system recovers. This issue is also relevant in the multi-site case, since the data stored at any individual site should survive the failure of the site; but the problems go far beyond this. In the single-site case, between the time of the failure of the system and its recovery, the system is

simply out of commission. However, in the multi-site case, the principal issue is enabling the remaining system sites to continue in operation while some sites are down. The difficulties in achieving this arise from the fact that site failures (and recoveries) will occur asynchronously with the operation of the system as a whole; it is hard to achieve smooth systemic operation as individual sites are going up and down in unpredictable ways. A useful perspective on this work is that we are concerned with developing techniques for the robust operation of a procedure being performed by a set of cooperating processors. Our work has been directed towards a specific distributed data base system, but we believe that our techniques are relevant to other data base systems and to distributed systems in general.

The purpose of the SDD-1 reliability mechanism is to guarantee that the redundant update strategy continues to perform correctly in the face of failures of TM's, DM's or communications facilities.

2.4.2 Principles of Design

In designing these mechanisms we have attempted to abide by a number of general principles. First, we needed to follow the premise, inherent in the SDD-1 design, that there is no central or primary site in the system and that each site should be able to function autonomously. Thus we avoided, for example, schemes that involve a consensus decision to determine whether or not some particular site has failed.

Simplicity of mechanism was another important criterion. Adhering to it allows us to have confidence that the reliability procedures will work correctly when invoked. It also simplifies the problems associated with failures occurring during the operation of the reliability procedures themselves.

We wished to minimize the overhead that the reliability subsystem incurs during periods of normal system operation. However, since any reliability mechanism entails some amount of overhead, we have sought to make the amount of this overhead parameterizable in terms of the degree of reliability offered. Thus a higher level of

reliability can be achieved by altering the reliablity parameters and consequently increasing the associated system overhead.

Finally, we have attempted to isolate the reliability procedures as much as possible from the rest of the system software. To this end, we have designed a subsystem called the reliable network. This is a body of software that interfaces the bulk of the SDD-1 system to the network communications facilities; it provides the following features:

1. The ability to guarantee eventual delivery of a message, even if the destination site is down at the time the message is sent.

2. The ability to reliably broadcast a message. That is, to send the same message to a number of sites and guarantee that either all or none of the sites receive the message (i.e., a failure in the midst of broadcasting the message will not result in the message having been delivered to some sites but not to others).

3. The ability to monitor foreign sites and execute appropriate routines in the event of their failure or recovery.

2.4.3 Types of Failure

Before we can examine how SDD-1 can be made sensitive to the failure of individual system components, we have to clarify our use of the word "failure". We can identify the following distinct forms of failure that may affect a system such as SDD-1, that consists of a collection of independent modules connected by means of a communication network.

1. Module failure. By this we mean that an individual component ceases function for some period of time, due to hardware or software failure. We assume that failure of a module can be detected by other modules that attempt to communicate with the failed one. We also assume that when the failure is repaired, the module will institute an appropriate recovery procedure. The component may fail in such a way that the failure does not affect the non-volatile storage maintained at the site, or the

crash may be catastrophic (e.g., scratching of a disk surface). The module may remain "down" permanently or it may cease to function for only a short (but arbitrary) period of time.

2. Module malfunction. This means that the module does not cease to operate, but continues in operation in an incorrect fashion. It is, of course, impossible to categorize all the ways in which a module may malfunction, but from the point of view of other modules in the system, we can identify the following four nodes of module malfunction:

   a. Failure to communicate. The module does not (and will not) supply any message to other modules that are waiting to hear from it; these expected messages may be in response to some stimulus or may be autonomously generated. This situation most likely results if the module is stuck in a loop, or if a process is left in a ready state, or if the routine responsible for dispatching messages has failed. Observe that this state is very difficult to detect through external observation; it is hard to

distinguish between this and a module that is merely operating slowly.

b. Slow to communicate. The module will eventually communicate with other modules but will take an unusually long time to do so. This state is difficult to distinguish from the preceding one.

c. Inappropriate communication. The module sends a message that is inappropriate in the given context. For example, a TM may be waiting to receive an acceptance or a rejection of a READ it sent a DM, but the DM instead replies that the computation is complete.

d. Incorrect communication. The module malfunctions and sends a message that is meaningful but is not the correct one in the circumstances. For example, a DM signalling a TM that it has rejected a READ when in fact it has accepted it.

3. Communication failure. Because of malfunction of the communications network connecting the components, a message being sent from one module to another may be lost, or garbled, or duplicated, or delayed, or delivered out of sequence with messages sent before or after it.

4. Network partition. Because of an extreme failure of the communication system, two modules needing to communicate with each other may be unable to establish a connection. In particular, a TM may be unable to reach a DM at which it seeks to read or update some variables or with which it needs to synchronize (e.g. to send an ALERT or wait for WRITE).

Our principal concern in this paper is with module failures, with modules that manifest malfunction by failing to communicate or by being very slow to do so, and with certain kinds of communication failures. We focus our attention on non-catastrophic module failures, which lose the contents of (volatile) memory but do not disturb (non-volatile) storage; the reason for this choice is that the problems of reducing a catastrophic failure of a

single-site system to a non-catastrophic level are well-known, and apply as well to a distributed system. Most communication failures are best detected and handled by the communications network; we shall be concerned with failures of the communications net that result in lost, delayed, or duplicated messages. We do not address the general problem of module malfunction, since there is no way, in general, for one module to determine that another is sending incorrect messages.

2.4.4 Goal of Reliability

Our goal is to develop mechanisms that will handle an arbitrary number of failures, so long as enough modules remain operating to provide full system function. Failures will of course in general occur asynchronously with the operation of the system as a whole, as will module recoveries. A module that has failed may remain in that state for an arbitrary amount of time; in particular, it may never recover.

Our concerns will include the detection of failures in a uniform and consistent fashion; the responses to be taken to the detection of a failure; and the actions to be taken at the time of recovery from the failure.

2.4.5 Reliability Problems

By analyzing the non-resilient TM and DM specifications presented earlier, we can determine that the module failure of a TM can have the following consequences:

1. If the TM fails while processing a transaction, but before sending out any of the WRITE messages, then the transaction will be left in an incomplete state. Since no WRITES have yet been sent out, this will not result in any database inconsistencies. Yet the DM's involved in the transaction will save their local workspaces indefinitely. This is an expense that should be avoided.

2. If the TM fails while sending out the WRITE messages of some transaction, then inconsistencies may result. That is, suppose the TM fails after sending out some but not all of the WRITEs. Then some DM's will receive the new values of the transaction's write variables, while others will not; this will cause an inconsistency between two

versions of the same variable or between related variables at distinct sites.

3. If a DM, in resolving a READ condition, is waiting to hear from the failed TM, then a transaction may be delayed indefinitely. Suppose that TM1 issues a READ to DM1 with the attached condition (TM2,t1); suppose further that TM2 failed at time t2, t2<t1. Then DM1 will not reject the READ; but it cannot accept it until it receives a WRITE from TM2 stamped later than t1. But that will not occur until some time after TM2 recovers. The problem is that DM1 can only accept the READ when it is sure that it has received all WRITEs from TM2 stamped before t1. If DM1 accepts the READ while TM2 is down, TM2 may then recover, run a transaction with its local clock still less than t1, and send DM1 a WRITE based on this transaction; this will invalidate DM1's earlier acceptance of the READ.

4. Other TM's may not be able to run certain transactions. As we have said, in order to run some transaction, TM1 may need to issue to TM2 an ALERT message, stamped at time t1; TM2 will reject this ALERT if it has already run a transaction

later than t1, and will eventually accept it otherwise. But suppose TM2 is down when TM1 issues the ALERT. Then TM1 has no way of knowing the time of the last transaction run by TM2. Therefore, TM1 will have to assume the worst and will not be able to run the transaction requiring the ALERT until TM recovers.

A similar analysis shows that the failure of a DM can have the following consequences which must be addressed.

1. Some of the data base may be lost from the system. The failed DM may contain the only copy of some variables in the entire system; the failure of the DM thus loses these variables and prevents any transactions that need to read them from running. This problem is beyond our purview; this is a real system failure, and no resiliency mechanism will serve to adjust the situation.

2. Some active transactions will not be able to run to completion. Any transaction in which the failed DM participates at the time of its failure will not be able to complete its execution.

3. Some transactions will not be able to access their read variables while the DM is down. Some transactions may normally obtain some of their read variables from the failed DM. If they attempt to read these variables elsewhere, there is the possibility that changes in the protocols will be required, both to these transactions and possible to others. This may result because protocol assignment is based on a global analysis of the interactions between transactions.

4. The copies of variables stored at the DM will become inconsistent with other copies of them stored elsewhere in the system. While a DM is down, a transaction may be run that updates a variable stored at the DM. Obviously, the DM cannot receive and process WRITE messages while it is down. So when the DM does recover, its local data base will be out of date.

If a TM does not fail, but malfunctions so that it gets into a state where it does not communicate with the rest of the system, the consequences are similar to those that can occur when it fails: If the malfunction occurs during the processing of a transaction, the transaction will be

left in an incomplete state. If the malfunction occurs while the TM is sending out WRITE messages, some will not be sent and inconsistencies may result. If a DM is waiting to hear from the malfunctioning TM, a transaction being run at another TM will not be able to run. If another TM issues an ALERT to the malfunctioning TM, no response will be received and the transaction requiring the ALERT will not be able to run.

If a DM enters a non-communicative state, any transactions in which it participates will not be able to complete, transactions that wish to read variables from it will not be able to run, and its local data base will become obsolete.

A TM or DM that is slow to communicate (perhaps because of a heavy system load on it) will cause the same difficulties as if it were in a non-communicating state. If a module's malfunction is manifested by inappropriate communication, the results are essentially the same as if it were not communicating; the significant point is that the messages for which other modules are waiting will not be forthcoming. Similarly, a lost, delayed, or garbled message (caused by communication failure) will have the same effect as though the sending module had failed prior to sending the message. (Because of the nature of the

protocols and the way timestamps control updating, no serious damage can result from a duplicated message.)

Thus, the effects of the different kinds of failures are similar, and are as we have described them above. However, each situation requires different techniques for detecting and coping with the failure.

2.4.6 Reliability Mechanisms

We will now examine some of the specific mechanisms which have been formulated to deal with these problems.

2.4.6.1 Failure of a Module Involved in Transaction Execution

The execution of a single transaction will span several modules within the system: the TM module supervising execution and any number of DM modules from which data is being read. We have decided that if any of these modules fail, then the transaction will be aborted (and perhaps resubmitted). While strategies which attempt to keep the transaction active can be conceived, we have chosen this approach in compliance with the goal of design simplicity.

It will be necessary for this strategy that modules be able to become aware of the failure of any of the other modules participating in a transactions execution. This is provided by the WATCH facility. A WATCH function at each module can be directed to detect the failure of another designated module. Because a module failure cannot be detected unless communication with the module is attempted, WATCH will periodically send a "PROBE" message to the foreign module. If the PROBE detects that the module is down, the software process requesting the WATCH will be notified. Of course, the PROBE message is obviated when a message has been recently received from the WATCHed site.

Rather than have every module participating in a transaction WATCH every other module, the following more economical procedure is used. The TM supervising the transaction WATCHes all the DM's involved, and each of the DMs WATCHs the TM. If the TM fails, then all of the DMs will abort the transaction in progress. If one of the DMs fails, the TM will detect this, abort the transaction locally, and send an ABORT message to the relevant DMs. Upon receipt of the ABORT message, the DMs will abort the transaction execution at their site.

2.4.6.2 Reformulation

A class which would normally read data from a failed DM may "reformulate" its READ requests to read alternate copies of the data from other DMs. Such a reformulation will, in general, require a change in the concurrency control protocol used by that transaction class. The question which must be answered is this: Is the change in protocol due to reformulation confined to that single transaction class, or do other transaction classes also have to alter their protocols? The problem is that if other transaction classes are affected, then mechanisms must be established to inform them of the reformulation.

While details of the analysis are beyond the scope of this report, we have found that protocol alteration is local to the transaction class which reformulates, so special mechanisms for reformulation communication are not needed.

2.4.6.3 Recovering WRITEs Issued while a DM was down

When a DM recovers, it must bring its local database into sync with those of other DMs in the system. This requires processing of WRITE messages it would have received had it not crashed. A first cut at a mechanism to solve this problem would be to have the sender simply buffer any WRITEs to failed DMs. When the failed site revives, it first retrieves and processes any buffered WRITEs before resuming normal operation. The difficulty with this approach is that the sending site may be down when the receiving site recovers.

The approach we have designed solves this problem through the use of "spoolers". A spooler is simply a message buffer at a site other than the receiver's. When a receiver is down, the sender will buffer messages in one or more spoolers (one of the spoolers will reside at the sending site). When the receiver recovers, it will retrieve its messages from one of the spoolers before resuming normal operation. Parameterizing the number of spoolers to be used gives us control over the degree of resiliency achieved by the system. A larger number of

spoolers increases the probability that at least one of them will be up when the receiver recovers. This higher resiliency, however, is bought at the cost of the added communication needed for the extra spoolers.

Details of the spooling mechanism are somewhat intricate and critical since the design of the spooling algorithm must take into account the spontaneous asynchronous failure and recovery of participating sites during the spooling and despooling phases. These design details are covered in [HAMMER and SHIPMAN 1978].

2.4.6.4 Resolving READ Conditions Against a Failed TM

In order to positively acknowledge a READ message, the receiving DM must ensure that the READ conditions attached to the READ message are satisfied (or satisfiable). A READ condition requires that all WRITE messages from a particular transaction class with timestamps less than some t have been received and processed by the DM, and that all WRITE messages from the transaction class with timestamps greater than t have not been processed. A problem occurs if the TM which runs that transaction class is down and messages through t have not yet been received. Potentially, the READ condition would have to remain unresolved until the TM recovered.

Fortunately, the spooler mechanism guarantees that when a site is down all messages sent from that site before it failed will have been received. Since the DM processing the READ condition knows that the TM in question is down, it can safely assume that all messages through t have been received since no additional transactions could have been executed at that TM beyond the one for which it has already received WRITEs.

The only additional requirement is that when the TM recovers, it must be sure to timestamp all new transactions with a timestamp greater than t. This is handled as part of a clock recovery mechanism. As part of a site's recovery procedure, it sets its own clock to a time at least as great as that of any other clock in the system. A DM will necessarily have a clock time as great as t if it positively acknowledged a READ condition with time t. So, the clock recovery procedure guarantees that when the TM recovers, all new timestamps will have a time greater than t.

2.4.6.5 Resolving ALERTs Against a Failed TM

The ALERT message used in P4 processing is issued to a TM. The TM positively acknowledges the ALERT if it has not processed any transactions with a timestamp greater than a given t, and it further guarantees that all new transactions do have timestamps greater than t.

Since the ALERT cannot be issued to a failed TM, the ALERTer needs to determine for itself whether or not the ALERT would be accepted. In order to do this, it must determine an upper bound on transaction timestamps assigned before the TM went down. If the ALERT time is less than this upper bound, the ALERTer will consider the ALERT as not acceptable.

To ensure that an upper bound can be safely arrived at, a "time signal" mechanism is used. The time signal is a message containing the current sender's clock time. Each TM site must send a time signal (and have it acknowledged) whenever its local clock advances DELTA since the last time signal or other timestamped message was sent to a site. Time signals need to be sent to every other TM site. With this mechanism in operation, it is only

necessary to add DELTA to the time of the last timestamped message to determine an upper bound on a TM's clock at the time of its failure.

The additional ALERT guarantee, that all new transactions after the ALERT have timestamps greater than the ALERT time, is met through the clock recovery procedure discussed under "Resolving READ conditions against a failed TM".

2.4.6.6 Failure of a TM During the WRITE Phase

As previously discussed, if a TM fails during the execution of a transaction, the entire transaction is aborted. After execution completes, however, results are sent to other DMs via WRITE messages. If the TM fails in the midst of sending WRITEs, a potentially dangerous situation arises, since some of the databases will have received the WRITEs and some won't have. The database will be inconsistent.

The solution to this problem is the use of a technique we call three-phase broadcast. With three-phase broadcast, the system guarantees that all of the WRITE messages will be received or that none of the WRITE messages will be

received. (Three-phase broadcast is a variant and extension of a mechanism called two-phase commit.)

During the first phase of the broadcast, the WRITE messages are sent to the appropriate DMs. The WRITE message includes a list of all other DMs to which WRITE messages for this transaction are being sent. The DMs do not process the WRITEs, however. They are held pending a COMMIT message from the TM. During the final phase, COMMITs are sent to all the DMs.

After a DM receives a WRITE message, it begins WATCHing the TM for failure. If the TM fails before a COMMIT is received from it, the DM must determine whether the TM was still in the first phase (in which case the transaction is aborted and the WRITE ignored) or had entered the final phase. It does this by inquiring of the other DMs which were to receive WRITE messages whether or not any had received COMMITs. If so, the TM had entered the final phase. If not, the TM had not entered the final phase.

If the DM fails after receiving a WRITE but before receiving a COMMIT, it must inquire upon its recovery whether or not COMMITs had been sent out.

A complication with this strategy arises when the only sites to receive a COMMIT fail immediately after receiving

it. In such a case, other DMs will not be able to determine that the TM had entered the final phase. To resolve this issue, a middle phase is added to the mechanism. During the middle phase, the TM sends n PRE-COMMIT messages to n other sites. The n PRE-COMMIT sites must be different from the first n sites to which COMMIT will be sent. The PRE-COMMIT sites are included in the list of sites attached to the WRITE messages, if not already included as COMMIT sites. A DM receiving a PRE-COMMIT will not process a pending WRITE, but will have positive knowledge that the TM has completed the first phase. With this additional middle phase, if all the sites to receive a COMMIT fail immediately after receiving it, the sites receiving the PRE-COMMITs will suffice to cause all the WRITEs to be processed. If a site fails after receiving a PRE-COMMIT, it must inquire upon its recovery whether or not the WRITE was actually considered to be COMMITed by the other sites.

Now, the only situations in which the three-phase broadcast mechanism does not work are those in which all of the PRE-COMMIT sites and all of the COMMIT sites which have received COMMITs fail after receiving their message. The probability of this happening can be made arbitrarily small be increasing the number of PRE-COMMIT sites.

2.4.6.7 Network Partitions

A network partition occurs whenever two sites are unable to communicate with each other. A network partition can occur in space (e.g., when communications failures split a network into two independent subnetworks) or in time (e.g., when all of the spoolers for a site are down at the time that the site recovers). Network partitions cause serious problems because it is not possible for sites to coordinate to ensure correct update synchronization.

"Syntactic" solutions to the network partition problem tend to involve overly severe restrictions on the sets of allowable update transactions In general, it appears that the best solution to any particular network partition situation requires a knowledge of the semantics of the database involved. Consequently, we believe that the most appropriate approach is for the system to detect network partitions and for a human database administrator to determine the correct responsive action to be taken.

3. NOSC Database Services

The other major ACCAT activity undertaken by CCA during the reporting period concerned itself with the installation and maintenance of Datacomputer and Datacomputer related software at NOSC. Specifically, two Datacomputer enhancements, precompiled requests and priority scheduling, were delivered, a boolean alerting facility was designed and implemented and new versions of the Datacomputer, TAP, DCSUBR and DCPKG were installed at NOSC. In addition, CCA provided consultation to NOSC on the use of software it delivered.

3.1 Precompiled Requests

In order to augment performance in the ACCAT environment, a facility was added to the Datacomputer to reduce the overhead of Datalanguage compilation. This facility allows common Datalanguage requests to be compiled once and stored in their compiled form. When the user wishes to run the request, he/she simply executes the precompiled request and avoids most of the compiler overhead. The initial version of precompiled requests was installed in September, 1977 and the second version has been implemented and will be installed at the end of January, 1978.

The initial version of precompiled requests gives the user the capability of compiling, storing, executing and deleting requests with the use of three new Datalanguage commands. This initial implementation saves precompiled requests for the duration of one Datacomputer session. The new Datalanguage commands are:

1. STORE <request-name> <DL-request>;

   This command causes the system to compile <DL-request> and save the result in such a way that it may be referred to simply as <request-name>.

2. EXECUTE <request-name>;

   This command causes the precompiled request, <request-name>, to be run.

3. DELETE <request-name>;

   DELETE deletes the precompiled request stored under <request-name>.

The second version of precompiled requests permanently stores the request in the Datacomputer directory. The CREATE command was modified to permit the creation of request nodes in in the following way:

CREATE <pathname> REQUEST [<DL-request>];

This creates a node in the directory for storing precompiled requests. If <DL-request> is supplied, it is compiled and stored in the node. The STORE command can be used to compile a request and store it in a previously created request node. As in the first implementation,

EXECUTE causes a stored request to run. DELETE deletes the request node from the Datacomputer's directory. The LIST command was modified to output useful information about precompiled requests. The type of the node is REQUEST and the %SOURCE option prints the Datalanguage associated with the request.

3.2 Initial Priority Scheduling

The Datacomputer priority feature enables different Datacomputer jobs to be assigned different priorities. This is an important capability in a command and control environment where vital queries must be able run rapidly without being hampered by other less critical applications.

Priorities may range anywhere from 1 to 1000. These levels are grouped into priority classes. Jobs in higher classes preempt jobs in lower classes.

Class 1	priority levels 1 through 800
Class 2	priority levels 801 through 900
Class 3	priority levels 901 through 1000

The initial priority capability provides a means for automatically assigning certain users a high priority upon

login and preventing users from setting their own priorities above certain limits. In this implementation, a user will cause all other users in lower priority classes to be suspended until he/she has finished running and conversely a user will not be able to start running until all users in higher priority classes have finished.

The priority limit can be established for a simple node (non FILE/PORT) by specifying the priority option (,P=<integer>) in the CREATE/MODIFY command. If a node has no limit specified, it is set by its superior node.

The initial default priority assigned to a user on login may be modified by using the Q option in the privilege block (,Q=<integer>) for the login node. This option is not passed down to subordinate nodes. When a user logs into a node that has the Q option specified, the job's priority is set to the Q value. If the Q value is greater than the priority limit, the job's priority is reduced to the priority limit. Once the user's priority has been established the following message is output:

;0033 <date/time> ASLOG: USER='<login pathname>', Q=<priority>

The PRIORITY command permits the user to change his/her priority up or down within the priority. Once again, the priority may not be raised above the user's limit.

3.3 Boolean Alert Feature

A boolean alerting feature was designed and implemented during the reporting period. It will be delivered to NOSC at the end of January, 1978. The alerting capabilities are embodied in the WATCH command.

A Datacomputer user may request that a file be watched for certain types of modifications by invoking the WATCH command. A watch may be removed from a file by using the UNWATCH command. A file may have at most 10 watchers.

When a request that causes modifications is run against any file having one or more outstanding watches on it, the system will determine which if any of the watches are satisfied. An alert is triggered for any of the watches that are satisfied. The triggering of an alert causes a message to be sent to the watcher indicating the file that was modified and the alert identifier (a unique identifier established when the WATCH command is executed.) The format of the message to the watcher is:

!A290 <date/time> ANAM5: ALERT/FILE = <alert-identifier> <pathname>

3.3.1 The Observe Privilege

In order to watch a file, the observe privilege must be granted. This privilege is granted and denied in the same way as are the data, control and login privileges. The letter "O" is used to specify the observe privilege.

3.3.2 The WATCH Command

The syntax of the watch command is:

WATCH <pathname> <options>;

<pathname> is the name of any Datacomputer file granting the observe privilege to the user. One or more options may be specified, each preceeded by a comma (see below for a detailed specification of the options.) If the WATCH is valid, the user will receive a message that includes the unique alert identifier. The message's format is:

!A287 <date/time> COWS: ALERT ID = <number>

The following possibilities exist for <options>:

a. ,A=[<host>,<socket>]

The A option specifies the address in terms of host and socket where the alert message will be sent. The A option is required unless the O option is specified (see below).

b. ,M=<list-of-modes>

The M option specifies the modes of modification of interest. One or more modes may be specified in any order. If no M option is present, a default of update mode is supplied by the system. The possible modes are:

W - write

A - append

U - update

R - remove

c. ,O=<pathname>[<output-containers>]

The O option requests that the system create an output file with the name <pathname> and append to it <output-containers> each time the alert is triggered.

The system will create <pathname>. It will include two virtual expressions, ALERT%ID (the alert identifier) and FILENAME (the watched file's name), a 12 character string DATE%TIME (date and time the alert was triggered) in addition to <output-containers>.

<output-containers> is a sequence of elementary field names (in the file being watched) separated by commas. Each field may be preceded by a prefix. The prefix "O:" indicates the old field value and "N:" indicates the new field value. The default prefix is N:. The O: prefix is valid for remove and/or update modes and the N: prefix is valid for write, append and/or update modes. The output file will have one field for every field in <output-containers>. The field names in the output file are created in the following manner:

O":<field-name> for O: prefixes

N":<field-name> for N: prefixes

The output file may be treated as any other Datacomputer file. If the O option is present without the A option, values will be appended to the output file but no alert messages will be sent.

d. ,U=[<updated-fields>]

This option is only valid if update mode is specified. It indicates that the alert should be triggered if any of the fields in <updated-fields> is modified. <updated-fields> is a list of field names separated by commas.

e. ,C=<boolean-condition>

The C option causes alerts to be triggered only if <boolean-condition> is true. <boolean-condition> is any valid Datalanguage boolean expression based on fields in the watched file and constants. The N: and O: prefixes may be used in the boolean expression. The default prefix is O: in this case. The prefix restrictions as described under the U option also apply here.

The <boolean-condition> will only be used to selectively trigger an alert if the entire boolean expression is legal with the mode of modification:

- REMOVE - boolean used only if it contains no N: prefixes

- UPDATE - always used
- WRITE/APPEND - boolean used only if it contains no O: prefixes

If both the C and U options are specified, the two conditions are ANDed to determine whether or not an alert has been triggered.

3.3.3 The UNWATCH Command

The UNWATCH command cancels a previously created watch. The user must have observe or control privilege on the file being watched. The syntax of the UNWATCH command is:

UNWATCH <pathname>,I=<alert-identifier>

The <alert-identifier> is the unique number returned by the WATCH command.

3.3.4 LIST Command Modification

The LIST command has a new option that enables a user with observe or control privileges for a file to see its outstanding watches. The new syntax is:

LIST <pathname> %WATCH;

The source for the WATCH command as specified by the watcher and the associated alert identifier (if the user requesting the LIST has control privileges) will be output for all outstanding watches on pathname.

3.4 Installations and Consultation

At the beginning of September, 1977, new versions of the Datacomputer, TAP, DCSUBR and DCPKG were installed at NOSC. CCA personnel were on-site to oversee these software deliveries. This version of the Datacomputer included the initial precompiled request facility and the initial priority scheduling feature. The TOPS20 Datacomputer which had been implemented in June of 1977 was also included in this delivery.

CCA personnel provided NOSC with consultation and debugging services throughout the reporting period.

4. Presentations and Publications

During the reporting period presentations describing elements of the SDD-1 design were presented at:

The ACM SIGMOD (Special Interest Group on Management of Data) annual conference

The Lake Arrowhead Workshop on Distributed Processing

The AIIE Conference on Distributed Processing

The Third International Conference on Very Large Databases

The First International Conference on Computer Software and Applications

In addition, the following technical reports and journal publications describing this design were produced:

CCA-77-09 The Concurrency Control Mechanism for SDD-1: A System for Distributed Databases (The General Case), P. Bernstein, D. Shipman, J. Rothnie, N. Goodman, December 15, 1977.

An Overview of Reliability Mechanisms for a Distributed Database System, M. Hammer, D. Shipman, Proceedings, 16th IEEE Computer

Society International Conference (COMPCON 78), San Fransisco, California, February, 1978.

A Survey of Research and Development in Distributed Database Management, Proceedings of the Third International Conference on Very Large Databases, Tokyo, Japan, October, 1977.

References

[ASTRAHAN et al]
Astrahan, M.M.; et al. "System R: Relational Approach to Database Management", ACM Transactions on Database Systems, Vol. 1 No. 2, June 1976, pp. 97-137.

[BERNSTEIN et al a]
Berstein, P.A.; Goodman, N; Rothnie, J.B.; and Papadimitriou, C.A. "Analysis of Serializability in SDD-1: A System for Distributed Databases (The Fully Redundant Case)", First International Conference on Computer Software and Applications (COMPSAC 77), IEEE Computer Society, Chicago Illinois, November 1977.
(Also available
from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139 as Technical Report No. CCA-77-05.)

[BERNSTEIN et al b]
Bernstein, P.A.; Rothnie, J.B.; Shipman, D.W.; and Goodman, N., The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The General Case), Technical Report No. CCA-77-09, Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139.

[CCA]
Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report, Technical Report No. CCA-77-06, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CODD]
Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", CACM, 13 (1970), pp. 377-387.

[HAMMER and SHIPMAN]
Hammer, M.M.; and Shipman, D.W., "Resiliency Mechanisms in SDD-1", Technical Report in progress. Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139.

[HELD et al]
Held, G.D.; Stonebraker, M.R.; and Wong, E., "INGRES -- A Relational Database System", Proceedings AFIPS National Computer Conference, Vol. 44, AFIPS Press, May 1975, pp. 409-416.

[PAPADIMITRIOU et al]
Papadimitriou, C.A.; Bernstein, P.A.; and Rothnie, J.B., "Some Computational Problems Related to Database Concurrency Control", Conference on Theoretical Computer Science, University of Waterloo, Waterloo Ontario, August 1977.

[REEVE et al]
Reeve, C.L.; Wong, E.; and Rothnie, J.B., Query Optimization Algorithm of SDD-1: A System for Distributed Databases, Technical Report No. CCA-77-15, Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, in progress.

[ROTHNIE et al]
Rothnie, J.B.; Goodman, N.; and Bernstein, P.A. "The Redundant Update Algorithm of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", First International Conference on Computer Software and Applications (COMPSAC 77), IEEE Computer Society, Chicago Illinois, November 1977.
(Also available
from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-02).

[ROTHNIE and GOODMAN a]
Rothnie, J.B.; and Goodman, N. "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.
(Also available
from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-04.)

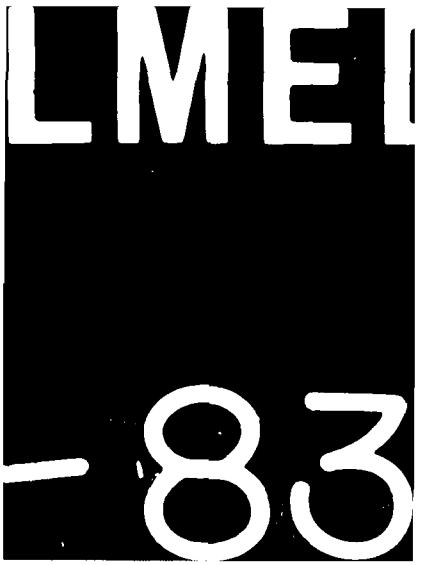
[ROTHNIE and GOODMAN b] Rothnie, J.B.; and Goodman, N. "A Survey of Research and Development in Distributed Databases Systems" presented at the Third International Conference on Very Large Data Bases, Tokyo, Japan, October 1977.

[STONEBRAKER et al]
Stonebraker, M.; Wong, E.; Kreps, B.; and Held, G. "The Design and Implementation of INGRES", ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976, pp. 189-272.

[THOMAS]
Thomas, R.H. "A Solution to the Update Problem for Multiple Copy Databases Which Uses Distributed Control", BBM Report No. 3340, Bolt Beranek and Newman Inc., Cambridge Massachusetts, July 1975.

[WONG and YOUSSEFI]
Wong, E.; and Youssefi, K. "Decomposition - A Strategy for Query Processing", ACM Transactions on Database[Systems, Vol. 1, No. 3, September 1976, pp. 223-241.

[WONG] Wong, E. "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.
(Also available
from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-03.)


LMEL
-83